Learn Version Control with Git

A step-by-step course for the complete beginner



目錄

Learn Version Control with Git 中文版

> 来源：[Learn Version Control with Git]

前言

关于专业

怎样才能称之为专业人员？是取决你对专业知识掌握的多少吗？是取决你对问题的理解能力吗？当然这些都是，但是这些都仅仅是片面的理解。

作为一个专业人员要学会使用一个正确的工具和培养一个良好的习惯。例如：一个五星级厨师不可能使用廉价的厨具来烹饪佳肴，因为他知道工欲善其事，必先利其器。同样的，一个顶尖的职业网球选手也不可能从来不进行耐力训练，因为他知道一个非常简单的道理，那就是打网球不仅仅只是击球过网这么简单。

同样道理，一个顶级的程序员，网络开发人员或网页设计师，他不可能从不使用版本控制，因为他知道在开发过程中犯错是在所难免的，他也知道在团队合作中如果不进行安全，简单有效的协作与沟通，而只是草草敷衍或者不紧密合作，在之后的开发阶段中将要花费更多的时间和精力。

不要介意去花费一点点时间来学习版本控制。它将帮助你在专业之路上迈出一大步。

关于这本书

学习这本书的目的是让你了解版本控制，并且尽可能快速简单的掌握 Git。但是和另外一些关于介绍版本控制的图书不一样，阅读这本书并不需要你有很专业的 IT 或者电脑背景知识，它也面向那些编程的初学者，软件构架师，或者是项目经理。在技术方面你也不需要有很多专业知识，我们会以循序渐进的方式帮助你来理解版本控制和掌握 Git。

话虽如此，Git和版本控制终归还是非常复杂且具有高度的技术性的。我不能保证非常全面的阐述其中的每一个原理，但是我将尽可能的向你解释和分析那些在实际应用中可能出现的案例，可能会被使用到的工作流程和相关的知识背景。

因为每个人都有他自己的专业背景，所以很难为每一个读者定义一个共同的出发点，出于这个原因，在本书的附录中我们会提供了一些基础知识的相关文档。

- 如果你不能确定是否要使用 Git 作为你的版本控制系统，你可以尝试阅读一下。为什么选择 Git？
- 如果你在考虑是否从 SVN 中解放出来，那么在这里 从 Subversion 过渡到 Git 你可以找到一些有用的帮助，它会给你列举出这两种版本控制系统间的差别。
- 如果你需要了解一些关于 Git 命令界面的操作，那么你一定要看看这个 命令 101。

祝你愉快的学习 Git！

Part 1 - 基础知识

什么是版本控制？

你可以把一个版本控制系统（缩写VCS）理解为一个“数据库”，在需要的时候，它可以帮你完整地保存一个项目的快照。当你需要查看一个之前的快照（称之为“版本”）时，版本控制系统可以显示出当前版本与上一个版本之间的所有改动的细节。



版本控制与项目的种类，使用的技术和基础框架并无关系：

- 无论是设计开发一个HTML网站或者是一个苹果应用，它的工作原理都是一样的。
- 你可以选择任何你喜欢的工具来工作，它并不关心你用什么样的文本编辑器，绘图程序，文件管理器或其他工具。

因此不要混淆版本控制的备份系统和一般的部署系统。当你开始尝试在你的项目中使用版本控制，你不需要替换和改变开发过程中使用的那些常用工具。

版本控制系统会记录所有对项目文件的更改。这就是版本控制，听起来很简单。

为什么要使用版本控制系统？

在开发项目中使用版本控制系统有很多好处。本章节将向你介绍其中的一些细节。

协同合作

试想一下，如果没有版本控制系统，当你需要处理那些共享文件夹中的文件时，你必须告知办公室里的所有人，你正在对哪些文件进行编辑；与此同时，其他人必须要避免与操作相同的文件。这是一个不现实和完全错误的流程。当你花了很长时间完成你的编辑后，可能这些文件早已经被团队里的其他开发成员修改或者删除了。

如果使用了版本控制系统，每一个团队成员都可以在任何时间对任何文件毫无顾虑的进行修改，版本控制系统可以把之后所有的改动合并成一个共同的版本，不论是一个文件还是整个项目。这个共同的中心平台就是我们的版本控制系统。

使用版本控制还有更多优点，这就要取决于你自己或者你的开发团队了。

版本存储（正确地）

经常性地保存项目的改动是一个非常重要的习惯。但是如果没有版本控制系统这个操作将是非常困难的，并且非常容易出错的：

- 你到底改动了什么？仅仅是针对一些特定文件的改动还是整个项目？首先你必须及时并小心地审查整个项目的每一个可能的改动细节，然后你需要付出大量且并不必要的时间来整理它。
- 你如何命名这些版本？如果你是一个思维很有条理的人，你也许会定义一个比较容易理解的通用命名规则（比如这样 “acme-inc-redesign-2013-11-12-v23”）。然而一旦涉及到一个多样性的改变（比如：在一次版本改动中，一些改动了标题区而另一些却没有被改动它），仅仅通过名字是很难追踪和判断这些改动的。
- 最重要的问题可能就是你如何知道在第一个版本和第二个版本之间到底进行了哪些改动？只有很少人会眞正地去花时间来仔细记录和观察每一个重要的变化，例如在项目文件夹的每一个README文件。

每一个版本控制系统仅仅对应一个项目。因此，在你的本地只存在一个版本，那就是这个项目的当前工作版本。除此之外，而其它所有之前的版本和改动都已经被有序地存储在版本控制系统中了。当你需要时，你可以随时来查看之前的任何一个版本，而且还可以得到整个项目的快照。

恢复之前的版本

要把一些文件恢复到上次改动之前的版本（甚至整个项目恢复到之前的版本）。这可能意味着你发现了一些严重的问题！如果你确定那些改动是错误的或者是没有必要的，那轻松的点几下你就可以简单地撤销它。在项目的每一个重要阶段，认识和正确地使用撤销这个功能会让你的工作变得非常轻松。

了解发生了什么

每当你提交一次对项目新的改动时，你的版本管理系统会要求你添加一个对这次改动的简短描述。除此之外（如果是一个代码或者文本文件），你还可以看到一个改动前和改动后的内容的详细对照。这样也可以帮助你很好地了解版本与版本之间的发展关系。

备份

备份是一个分布式版本控制系统（例如 Git）提供的非常好的附带功能。每一个团队成员都会在他的本地有一个完整的项目副本，包括整个项目的历史记录。如果你所依赖的服务器宕机了，或者是你的存储硬盘坏，所有你需要的恢复文件都可以在另外的团队成员的 Git 本地仓库中得到。

准备工作

命令行界面还是图形界面？

Git 提供两种主要的工作环境：“命令行界面” 或者一个 “GUI” 应用程序。使用哪一种界面都无所谓正确或错误。

在一方面，使用 GUI 应用程序会让你更有效和方便地使用一些复杂或者先进的功能。但这些复杂操作在命令行界面就显得过于复杂了。

在另一方面，我还是建议你首先来学习 Git 的命令，这样可以帮助你更进一步地了解一些重要的基础工作原理，而且不需要倚赖于任何一个图形界面的帮助。

参考

如果你完全不懂那些命令行，我准备了 “命令 101” 这个附录会提供给你一些最重要的基础知识。

当你已经具备了一定的基础知识后，你应该考虑一下使用一个具有图形界面的应用程序，使得你每一天的开发工作变得更轻松，更有效。 Windows 用户可能需要看看这个 **Tortoise Git**，而Mac OS的用户应该看看这个 **Tower**。你也会在这本书中找到一些图形界面应用程序的更为详细的介绍 (see "Part 5: 工具与服务")。

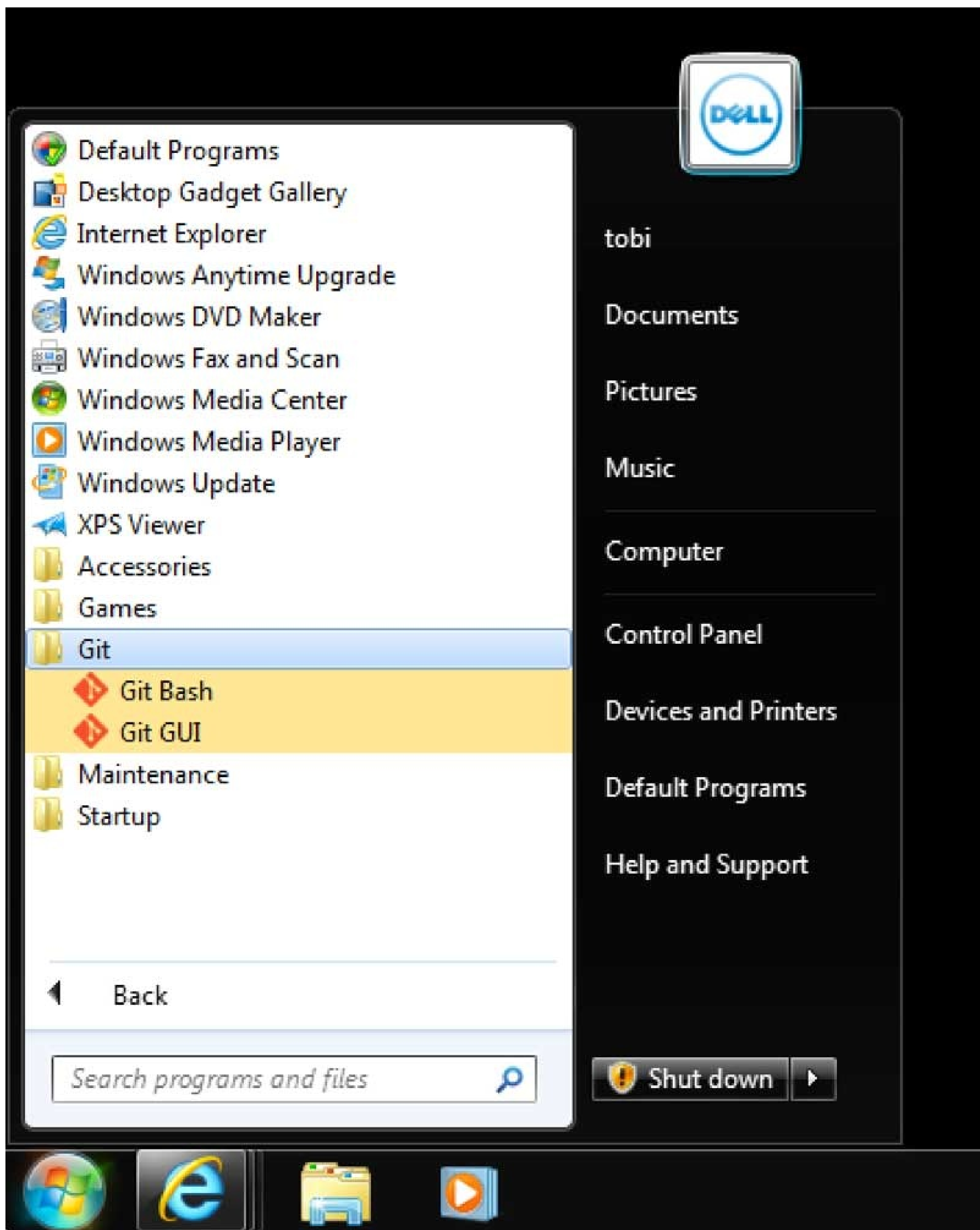
在你的计算机上安装 Git

安装 Git 是非常简单的，并且面向 Mac 和 Windows 用户还提供了一键安装方式。

在 Windows 上安装 Git

在 Windows 电脑上 “msysgit” 是一个非常强大的 Git 工具包。你可以从下面给出的地址下载它： msysgit.github.io

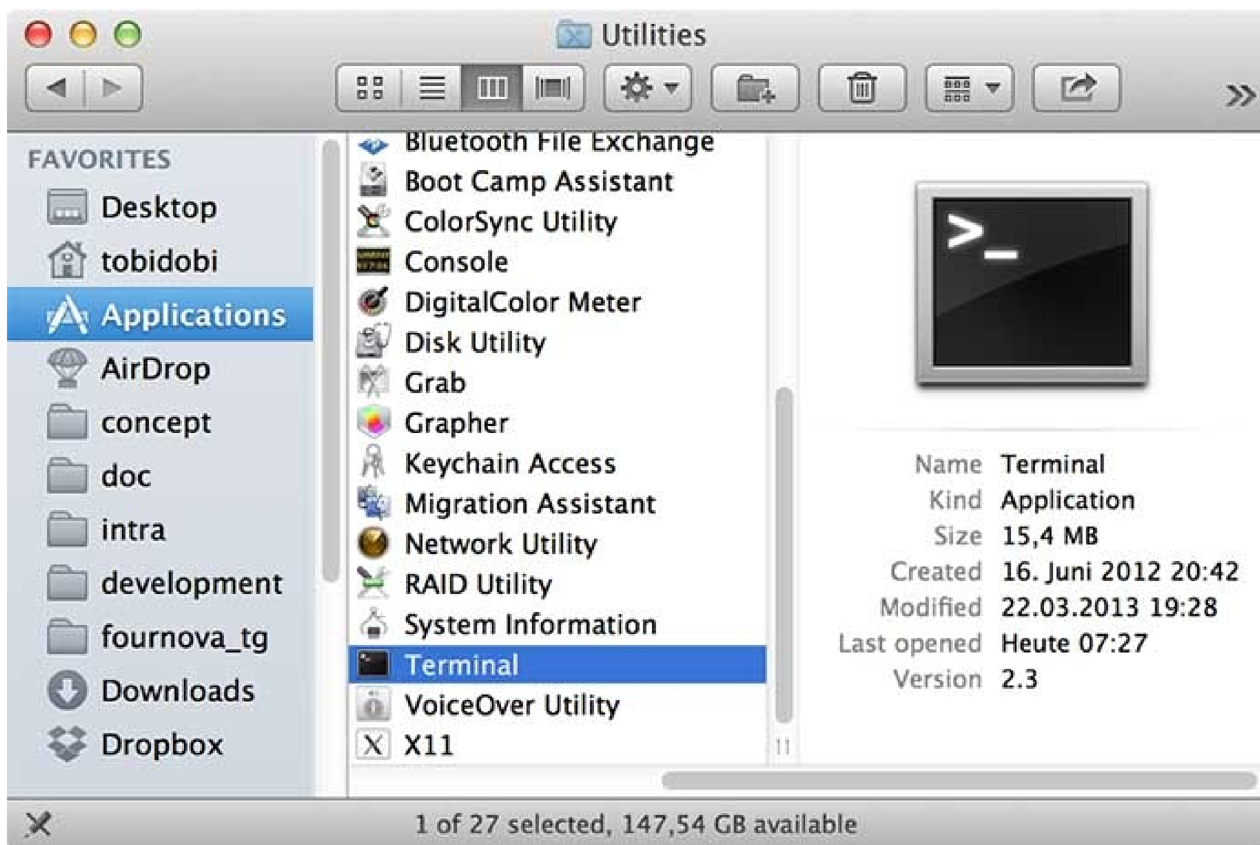
双击安装文件，在安装的过程中你可以一直选择默认设置，一直点击继续直到安装结束。当安装完成后你就可以使用 “Git Bash” 这个应用开始工作了。你可以在 “Windows” 开始菜单里的 “Git”目录找到这个应用。



在 Mac OS 上安装

在 Mac OS X 中，你可以从下面的地址下载它，然后单击进行安装： code.google.com/p/git-osx-installer/downloads/list?can=3

一但安装结束，你可以通过 Mac 提供的 “Terminal.app” 应用直接打开它。你也可以在你的 "Applications" 目录中的子目录 "Utilities" 中找到这个应用：



设置 Git

在打开Git之前，你需要完成一些最基本的设置。例如你的用户名，你的邮箱地址以及在命令行界面中的一些重要的显示设置：

```
$ git config --global user.name "John Doe"
$ git config --global user.email "john@doe.org"
$ git config --global color.ui auto
```

注释

和其他的专业书籍一样，在这本书中 “$” 符号就代表着这是一个在命令行界面中的命令。（输入命令时你不需要键入这个符号！）。因此，任何时候你一旦看到一行以 “$” 符号开始的代码，这就表示它是一个在 “Terminal” or “Git Bash” 里能被执行的命令。

除此之外，我建议你把这个 命令速查表 放在你的桌面上。这样你就不需要把每一个命令都记在心里了。

版本控制的基本工作流程

在我们开始搞明白 Git 命令之前，你有必要先了解一下版本控制的基本流程。这本书会向你一步步地详细阐述各种不同的工作流程。但是首先还是让我们先来一起了解一下版本控制的最基本的流程。

版本控制中的最基本的模块就是“仓库（Repository）”。

名词解释

仓库（**Repository**）

你可以把一个仓库想象成一个数据库，在那里你的版本控制系统存储了项目积攒的所有版本和元数据（metadata）。在 Git 中，仓库只是一个在你项目的根目录下名为 “.git” 的隐藏文件夹。你只要知道有这个文件夹存在就足够了。你没有必要也不需要去接触或是搞明白这个文件夹。

你有两种方法可以来获取一个本地仓库到你的计算机上：

- (a) 如果在本地计算机中已经存在了一个项目，但是尚未纳入版本控制系统中，你可以为这个项目初始化一个新的仓库。
- (b) 如果你需要得到一个已经纳入版本控制系统中的项目，而且它已经存放在一个远程服务器中 （比如在互联网或者是你的局域网中）。你只需要知道这个仓库的 URL 地址，然后克隆 （下载/拷贝） 到你的本地计算机中就可以了。

(1) 只要你有了一个本地仓库，你就可以在任何一个你常用的应用程序中 （例如：你最经常使用的编辑器，文件浏览器……） 开始你的工作了：修改、删除、添加、拷贝、重命名或者是移动文件。在这里你不须要考虑任何其它的事情，完成项目所要求的改动就行了。

(2) 当你已经确定完成了这次改动或者操作，你就需要开始来再次考虑版本控制了。因为是时候打包并提交你的这些改动了。

名词解释

提交（**Commit**）

一次提交包含了一组特定的变化。提交的作者必须为这个变化做一个简短的 "注释（commit message）"。这将有助于别人或者改动者本人在以后的时间里很好地了解和明白这次提交的意图，以及什么时间做了什么改动。

对于每组这样的更改，版本控制系统都会为你的项目缺省的创建一个新的版本。也就是说，会为每次提交创建一个单独的版本。这就是项目在这个特定时间点的快照（snapshot），版本控制系统会很有效的把这信息记录下来，而不是简单地对整个项目进行一次备份。并且提

交也能非常准确的知道你的文件以及目录的状态和关系，并对它们进行有效的操作，例如：恢复项目到某些状态。

(3) 然而在提交之前，如果你需要一个到目前这个时间点的详细改动概况，那么在 Git 中，你可以使用 “status” 命令来得到一个与上一次提交对比之后的改动列表，比如哪些文件你做了改动，是否新建了一些新的文件，或者删除了一些已存在的文件。

(4) 下一步，你需要告诉 Git ，在本地项目中你想打包哪些被改动的文件到下次的提交中。一个文件被改动了之后，并不意味着它会被自动打包在下次提交中。相反，你必须明确地指出你需要打包哪些文件。这就是说你必须要把它们添加到所谓的 “暂存区（Staging Area）”。

(5) 现在，在暂存区（Staging Area）中已经存在了一些改动过的文件，现在是时间真正的提交它们了。但是在这之前你还必须为它添加一个既简短又明确的注释（commit message），用来描述这次改动到底做了什么。提交完成后本地仓库会被更新，然后为这个项目建立一个新的版本。

(6) 如果你想要查看一下整个项目的开发状况（特别是处于团队开发中），使用 "log" 命令就能看到按时间顺序列出的所有提交的改动。这时你就可以看到有哪些改动以及那些改动的一些细节，从而更好的帮助你了解你的项目的发展状况。

(7) 此外，当你处于团队协作开发时，你需要和其它的开发团队成员共享你的改动，同时也想了解其它成员所作的改动，这时一个在服务端的远程仓库（remote repository） 就可以用来进行这些数据交换。

名词解释

本地和远程仓库（**Local & Remote Repositories**）

两种不同的仓库类型：

- "本地" 仓库（local repository） 是放置在你的本地计算机中的，它以一个隐藏文件夹的形式存储在项目的根目录（root folder） 中。你是唯一一个有权通过提交改动来操作这个仓库的用户。
- "远程" 仓库（remote" repository）则相反，它通常是位于一个远程服务器上的，比如在互联网上或在你的局域网络上。没有任何工作文件与远程仓库相关联：它没有工作目录（working directory），而是完全由 “.git”仓库目录组成的。开发团队使用远程仓库进行数据共享和交换，远程仓库是协作开发时的一个共同基础，每个项目成员都可以发布自己的改动，同样也都可以接收到其他成员的改动。

从一个未被纳入版本控制的项目开始

你现在正在着手在一个项目开发上，但是它还未被纳入版本控制系统的管理中，那就让我们从这个项目开始吧！在命令行界面中跳转到这个项目的根目录（root folder），然后键入 **“git init”** 命令来建立一个 Git 项目：

```
$ cd path/to/project/folder
$ git init
```

现在我们来看看在这个目录下都有哪些文件（也包括所有的隐藏文件）：

```
$ ls -la
```

你将会看到其中有一个新建的并且名字为 “.git” 的隐藏目录。为什么有这么一个新的目录呢？其实这是 Git 给我们创建的一个空的本地仓库（local repository）。为什么是 “空” 呢？开始时 Git 并不会自动地把当前项目中的所有的内容当作 “初始版本（initial version）” 添加到本地仓库中的，因此现在这个本地仓库还不包含任何你的项目文件。

名词解释

工作副本（**Working Copy**）

项目的根目录我们通常称之为 “工作副本” 或者叫做 “工作目录（working directory）”。这个文件夹包括项目所有的内容并且存放在你的本地计算机中。 你需要经常的询问版本控制系统来同步你的工作副本。一定要记住，在你的本地计算机中只存在唯一一个和项目版本所对应的工作副本，多重并行的工作副本是不被允许的。

忽略文件

通常情况下在很多项目或者开发平台中都会有一些你并不想要纳入版本控制系统的文件，例如在 Mac OS 中，哪些 “.DS_Store” 文件，它并不需要纳入版本控制中去。在其他的项目里，通常也会存在一些编译文件和临时的缓存文件。把它们纳入版本控制系统中其实是毫无意义的。因此当你使用版本控制时，你就必须决定哪些项目文件需要被纳入版本控制系统中，而哪些不需要。

概念

哪些文件不需要纳入版本控制中？

哪些文件需要被忽略呢？一个最简单分辨方法就是，那些在你开发项目过程中自动生成的文件。例如，临时文件，日志和缓存文件等等。 还有其他的例子，比如那些为编译代码所提供的密码或者个人设置文件。

下面这个连接：github.com/github/gitignore 可以帮助你更好地了解在不同的项目和开发平台上哪些内容不需要纳入版本控制中去。

忽略文件列表被存放在项目根目录中一个被称之为 “.gitignore” 的文件中。在这里强烈的建议你，在做第一次提交之前，首先应该定义好这个忽略文件列表。因为一旦这些文件被提交了，要想再次把它们清除出版本控制系统就很麻烦。

现在，让我们使用你最常用的编辑器来建立一个空的文件，命名为“.gitignore”，然后保存在项目的根目录下。如果你是在 Mac OS 中，你就可以直接在这个文件中加上如下的内容：

```
.DS_Store
```

如果还想忽略其它文件，很简单，在文件中为每一个需要忽略的内容添加一行。你也可以定义一个非常复杂的规则。 为了简单起见，在这个理列出一些在你定义忽略文件时最经常用到的例子：

- 忽略一个特定的文件：给出从项目根目录开始的路径和文件名，例如：path/to/file.ext 。
- 忽略项目下所有这个名字的文件：只要给出文件的全名，不要包括任何路径，例如 filename.ext 。
- 忽略项目下所有这个类型的文件：例如 *.ext 。
- 忽略一个特定目录下的所用文件：例如 path/to/folder/* 。

完成你的第一次提交

当你定义好了你的忽略规则后，现在是时间来为你的项目进行第一次提交了。在本书之后的章节里我们将继续深入细致地探讨提交的工作原理和详细流程。就现在而言，只须要执行如下的命令就可以了：

```
$ git add -A
$ git commit -m "Initial commit"
```

从一个已被纳入版本控制的项目开始

如果你要得到一个存放在远程服务器中的并且已经被纳入版本控制系统中的项目，你只需要给出项目远程仓库（remote repository） 的 URL 地址。这个 URL 可能采用如下形式：

- ssh://user@server/git-repo.git
- user@server:git-repo.git
- http://example.com/git-repo.git
- https://example.com/git-repo.git
- git://example.com/git-repo.git

URL 的形式并不重要，你只需要在命令行界面中键入 “git clone” 命令和URL。当然，你必须明确知道你要下载到你本地计算机的那个位置，首先跳转到这个目录中：

```
$ cd your/development/folder/
$ git clone https://github.com/gittower/git-crash-course.git
```

注释

你不需要自己去准备一个远程仓库，也无需把它克隆到你的本地计算机上，使用上面给出的远程仓库的 URL 就可以了。在本书接下来的部分将会使用这个仓库 （repository） 作为例子。

接下来 Git 将会完整地克隆这个仓库到你的本地计算机上，克隆完成后你就可以访问这个仓库了。

- 对于 “http” and “git” 协议，你不需要任何权限。
- 对于 “https” URLs，你可能需要输入用户名以及密码。
- 对于 “ssh” URLs （无论是以 “ssh://” 开头的或者是一个简易格式，“user@server...”），你必须要通过一个 “SSH 公钥（SSH Public Key）” 的认证。虽然它需要你做一点点额外的设置工作，但是这是一个被广泛使用的安全有效的模式。在本书稍后的章节中会专门来介绍这个方面的详细内容。“使用 SSH 公钥验证”

工作在你的项目上

无论是建立一个全新的本地仓库（local repository）或是克隆一个远程仓库（remote repository）到本地计算机，这两者并没有多大的区别。现在在你的计算机上已经拥有了一个 Git 本地仓库了。这也就意味着你可以在这个项目上开始你的工作了，使用任何一个你常用的编辑器来完成对项目文件的修改、创建、删除、移动、拷贝或者是重命名。

概念

文件的状态

一般情况下在 Git 中文件有两种状态：

- 未被追踪的文件（**untracked**）：如果这个文件还未被纳入版本控制系统中，我们称之为“未被追踪的文件”。这就表示版本控制系统不能监视或者追踪它的改动。一般情况下未被追踪的文件会是那些新建的文件，或者是那些没有被纳入版本控制系统中的忽略文件。
- 已追踪的文件（**tracked**）：所有那些已经被纳入版本控制系统的项目文件我们称之为“已被追踪的文件”。 Git 会监视它的任何改动，并且你可以提交或放弃对它修改。

暂存区（**Staging Area**）

当你完成了对项目文件的改动后，你想要保存这个改动到你的项目中去。换句话说，你想要提交你在这些已追踪文件 （tracked files）上的改动。

黄金法则

#1: 一次提交只对映一个相关的改动

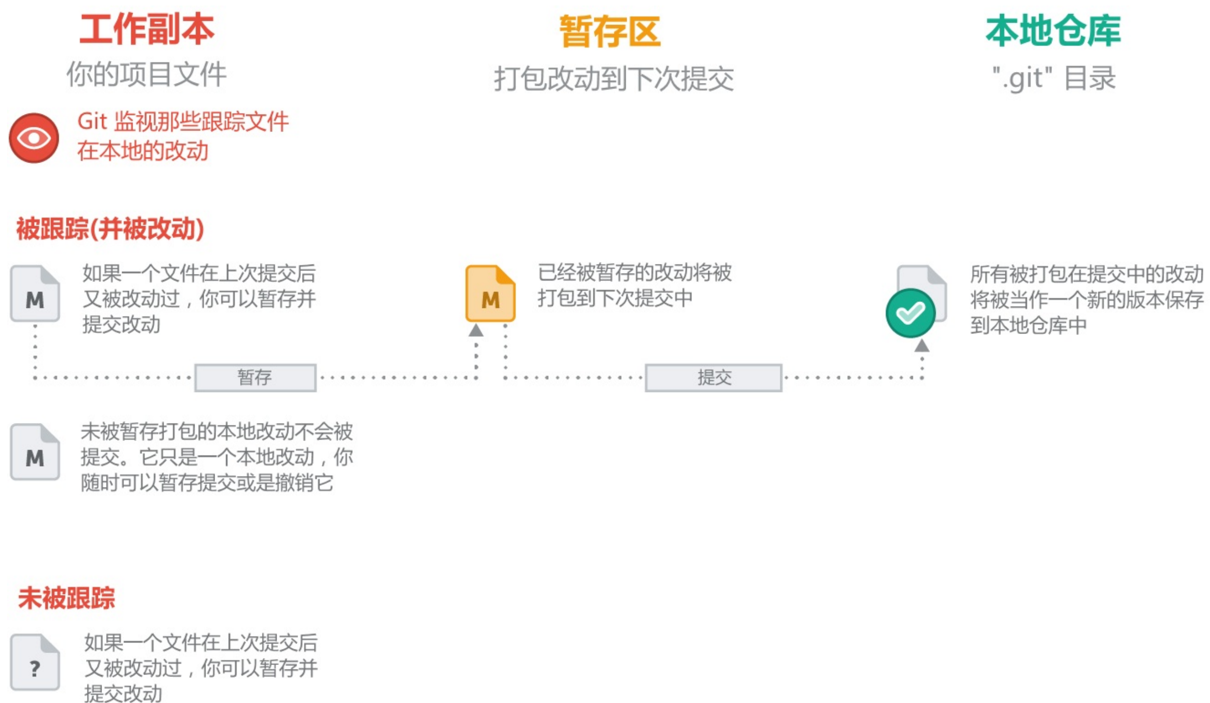
当你想要进行提交时，非常重要的一点就是，它只能包含一个相关的改动。也就是说，不要在一次提交中包含一些和提交目的无关的改动，每一种不同的改动要打包在不同的提交中。比如，在一次提交中既添加了一个新的功能（例如：用户注册功能），又包括了对一些错误的修复（例如：修复了Bug#122）：

- 理解这些复杂的提交对于开发团队里的其他成员来说是非常困难的。可能过了一段时间连你自己也糊涂了。你的队友不得不先要搞明白这个错误，弄清楚你为什么要同时修复它，之后才能试图了解这个新添加的用户注册功能。
- 不可能复原或者撤销其中一个改动。试想一下，那个新添加的用户注册功能可能存在很严重的错误，你想撤销它但是你又不想撤销那个修复错误的改动。你该怎么办呢？

也就是说，一次提交一定要包括一个且仅仅只能包括一个和提交目的相对映的改动：修复两个错误（最起码的）你要进行两次不同的提交。或者当开发一个新的非常庞大的功能时，你必须要把它分成几个小的并且在逻辑上有意义的提交，这样做可以有效地减少产生错误的可能性。 这种小的提交可以让开发团队的其他成员更好地理解这个改动。一旦发现这个改动有问题，就可以非常简单地撤销，而不会影响到其它的功能。

然而，当你专注于在你的项目开发中时，你不可能保证你的每次改动都只对应一个目的，因为你总是在多个方面同时工作的。

这就须要引入一个新的概念“暂存区（Staging Area）”，这是一个 Git 中最强大的功能，而且也非常好用。它可以让你来确定哪些修改将被打包在下次提交中。因为在 Git 中并不是简单的把每一个改动后的文件都自动打包起来的。相反，每一次提交打包都要“手动完成”，哪些改动需要被打包在下次提交中都需要你自己决定，是否需要“添加到暂存区域”或者简单的说“被暂存”。



得到所有的改动的详情

让我们来看看到目前为止都已经完成了什么。为了得到你在上一次提交后的所有改动详情，你可以简单地使用这个命令“git status”：

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>... " to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#    directory)
#
#       modified:   css/about.css
#       modified:   css/general.css
#       deleted:    error.html
#       modified:   imprint.html
#       modified:   index.html
#
# Untracked files:
#    (use "git add <file>..." to include in what will be committed)
#       new-page.html
no changes added to commit (use "git add" and/or "git commit -a")
```

值得庆幸的是，Git 给我们提供了一个相当详细的总结，其中包括了三个主类别：

- “Changes not staged for commit” 改动的文件，但没有打包在下次提交中。
- “Changes to be committed” 改动的文件，并且已打包在下次提交中。
- “Untracked files” 未被追踪的文件。

准备开始提交

现在是时候来提交那些改动到暂存区（Staging Area）里去了，使用 “git add” 命令：

```
$ git add new-page.html index.html css/*
```

以上命名表示，我们添加一个新的文件 “new-page.html”，并且提交对文件 “index.html” 和在目录 “css” 下所有的改动到暂存区。如果我们想把文件 “error.html” 从版本控制中移除掉，我们必须使用 “git rm” 来完成它：

```
$ git rm error.html
```

让我们再次使用 “git status” 命令来查看一下详情列表：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   css/about.css
#       modified:   css/general.css
#       deleted:    error.html
#       modified:   index.html
#       new file:   new-page.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#    directory)
#
#       modified:   imprint.html
#
```

假设对文件 “imprint.html” 的改动和其他的改动没有任何联系，我们有意地不把这个文件添加到暂存区域中，那就表示，它不会被打包在下次提交中。对它来说现在仅仅是一个本地修改。我们可以继续对它进行工作，可能在以后的某个时间再去提交它。

提交你的工作

有了精心准备好的暂存区后，在提交之前还剩下一件事，那就是注释这次提交。

黄金法则

#2: 高质量的提交注释

花一点时间写一个好的提交注释是非常值得的，这样可以让开发团队的其他成员非常容易地明白你做这次提交的目的和你的改动（过了一段时间对你自己也有帮助）。针对你的改动写一个简短的注释标题（原则上不要超过50个字符），然后使用一个空行来分隔注释的内容。注释的内容要尽可能的详细并且要能回答以下几个问题：为什么要做这次修改？与上一个版本相比你到底改动了什么？

“git commit” 命令将提交你的修改：

```
$ git commit -m "Implement the new login box"
```

如果你有一个很长的提交注释，并且注释中包含很多段落，那你就不需要使用 “-m” 参数了，Git 会为你打开一个文本编辑器（具体打开哪个文本编辑器，你可以在 “core.editor” 设置它）。

概念

什么才是一个好的提交

一个高质量的手动提交对你的项目和你自己是非常有意义的。什么才是一个好的提交呢？在这里有一些基本的原则：

1. 提交要仅仅对应一个相关的改动 首先，一次提交应该仅仅只对应一个相关的改动。不要把那些互相毫无关联的改动打包在同一次提交中。如果这次提交出现了什么问题，解决和撤销它将是非常困难的。
2. 完整的提交 千万不要提交那些没有完全完成的改动。如果你想要临时保存一下你当前的工作，例如一个类似于剪贴板（clipboard）的功能，你可以使用 Git 提供的 “Stash” 功能（这个功能我们将在本书之后的章节为大家介绍）。但是一定不要直接提交它。
3. 提交前测试 当你提交你的改动时，不要理所当然地认为你的改动永远正确。在你提交你的改动到你的仓库前，进行有效的测试是非常重要的。
4. 高质量的提交注释 一次高质量的提交需要一个好注释。请参考本章之前的章节 “高质量的提交注释”。

最后，你须要养成一个频繁地进行提交的习惯。这样做将自然而然的让你避免一个很庞大的提交，并且使这些提交可以更好只对映一个相关的改动。

检查的提交历史

Git 会记录对项目改动的所有提交。特别是当与其他团队开发人员协同开发时。及时地查看那些最新的提交并理解它是非常重要的。

注释

在本书之后的章节 “通过远程仓库共享工作”，我们将向你详细介绍如何同团队的其他开发人员进行数据交换。

显示在项目中所有的提交历史记录可以使用 “git log” 命令：

```
$ git log
```

所有在这个项目中的提交将按时间顺序显示出来，最新的提交会出现在最上面。如果说所有的提交历史记录不能完全显示在一个屏幕中，在命名行界面的最下方会显示一个冒号（“:”）。这时你可以使用空格键来跳转到下一页，或是使用 “q” 键退出这个界面。

```
commit 2dfe283e6c81ca48d6edc1574b1f2d4d84ae7fa1
Author: Tobias Günther <support@learn-git.com>
Date: Fri Jul 26 10:52:04 2013 +0200

    Implement the new login box

commit 2b504bee4083a20e0ef1e037eea0bd913a4d56b6
Author: Tobias Günther <support@learn-git.com>
Date: Fri Jul 26 10:05:48 2013 +0200

    Change headlines for about and imprint

commit 0023cdddf42d916bd7e3d0a279c1f36bfc8a051b
Author: Tobias Günther <support@learn-git.com>
Date: Fri Jul 26 10:04:16 2013 +0200

    Add simple robots.txt
```

每个提交记录都包括（除其他事项外）如下的元数据（metadata）：

- Commit Hash
- Author Name & Email （提交人的姓名和电子邮箱）
- Date （日期）
- Commit Message （提交注释）

名词解释

The Commit Hash

每一个提交都拥有一个唯一的 ID ：这个40位字符的校验码我们称之为 “commit hash”。在那些集中式的版本控制系统中（例如： Subversion， CVS...）会使用一个依次递加的版本号码。这样虽然简单，但是它却不能被使用在 Git 这样的分布式版本控制系统中。原因就是在于，在 Git 中每个人的工作都是并行的，改动的 提交 都是 离线的 ，也就是说提交时并不需要连接到远程仓库。在这种情况下，远程仓库就不能确定哪个是第5次提交， 而哪个是在它之后的第6次提交。

在大多数项目中，这个哈希编码的前七位字符就已经能足够代表一个唯一的提交ID了，一般我们都会用这个简短的7位 ID 来代表一个提交。

除了这些元数据（metadata），Git 也可以显示出提交的更详细的内容。在 “git log” 命令后使用 “-p” 参数来显示一些更多的提交记录详情：

```
$ git log -p
commit 2dfe283e6c81ca48d6edc1574b1f2d4d84ae7fa1
Author: Tobias Günther <support@learn-git.com>
Date: Fri Jul 26 10:52:04 2013 +0200

    Implement the new login box

diff --git a/css/about.css b/css/about.css
index e69de29..4b5800f 100644
--- a/css/about.css
+++ b/css/about.css
@@ -0,0 +1,2 @@
+h1 {
+  line-height:30px; }
\ No newline at end of file
di.ff --git a/css/general.css b/css/general.css
index a3b8935..d472b7f 100644
--- a/css/general.css
+++ b/css/general.css
@@ -21,7 +21,8 @@ body {

 h1, h2, h3, h4, h5 {
   color:#ffd84c;
-  font-family: "Trebuchet MS", "Trebuchet"; }
+  font-family: "Trebuchet MS", "Trebuchet";
+  margin-bottom:0px; }

 p {
   margin-bottom:6px;}
diff --git a/error.html b/error.html
deleted file mode 100644
index 78alc33..0000000
--- a/error.html
+++ /dev/null
@@ -1,43 +0,0 @@
- <html>
-
-   <head>
-     <title>Tower :: Imprint</title>
-     <link rel="shortcut icon" href="img/favicon.ico" />
-     <link type="text/css" href="css/general.css" />
-   </head>
-
```

在本书之后的章节 "使用diffs检查改动的详情", 我们将会来一起学习这些输出内容的具体含义。

值得庆祝的时刻

祝贺你！你刚刚掌握了 Git 版本控制系统中的最基本步骤！休息一下，在我们开始下一步的学习之前先来干杯啤酒。

Part 2 - 分支与合并

分支可以改变你的生命

这个标题的确有点让人震撼，不过，至少我是这样认为的。事实上这并不夸张，有效地使用分支功能（branching）真的可以帮助你改善每一天的工作，让你成为一个更专业的程序员或者设计师。

首先，如果你已经拥有了一些在其他版本控制系统上的使用经验，那么我诚恳地请求你忘记那些你以前对分支（branching）和合并（merging）的认识。其实， Git 并没有做出什么新的发明，和其他很多版本控制系统一样都使用了分支这个理念。然而， Git 的分支概念在这个领域中的确是独一无二的，特别是在易用性和效率方面。

现在，让我们来看看为什么分支功能是如此重要。

工作在不同的背景中

在每一个项目中，我们要完成的所有工作都拥有有它不同的主题，并且它们都处在一个特定的上下文（context）背景环境下。开发每个功能，修复每一个错误，进行每一个设计上的尝试，甚至于你所开发的产品自身它们都应该被看作一个个独立的主题，并且在它所处的上下文环境中，这个主题与其他主题都毫无关系。 这样就有了在每个上下文环境中所对应的独立主题。但是，最起码你至少应该拥有一个最主要的主题，那就是你要给客户开发的产品本身，其次就是一些其他的开发主题，比如添加新的功能，修复错误，改动尝试等等。

在真实的项目中，所有的开发工作总是同时进行的，并且其每一个主题都处于一个特定的上下文环境下：

- 当你为了完成一个新的用户需求，你对这个页面准备了2套设计方案（主题 1，主题 2）……
- 同时你也想尝试的去修复一些错误（主题 3）。
- 另一方面，你还想升级一下你的 FAQ 页面（主题 4），或者……
- 团队中的另外一些开发人员正在为在线购物车添加一些新的功能（主题 5）……
- 还有其他一些同事正在尝试开发一种新的用户登录功能（主题 6）。

一个没有分支的世界

如果不清楚地区分在不同工作主题中每一个特定的上下文环境，那么就会产生一系列的问题：

- 如果你的客户只喜欢第二种设计方案，但是在你完成这个方案的同时，另外那个方案的改动也被实施了，那么你该如何保留你的这个有用的改动，并且撤销掉那些没有被采纳方案的改动呢？

- 如果客户改变了他的需求，我们不得不把那个为在线购物车添加的新功能删除掉，我们要如何清除这些代码呢？
- 如果那个新的用户登录功能已被证明不能被运用到真正的产品中去，而可能在这时它已经夹杂着很多其它不相干的改动了，那么我们该如何把这个功能剔除出我们的项目呢？
- 在不影响到其他的团队开发人员和开发主题的同时，你该如何去跟踪你的改动呢？

当你尝试着在一个单一上下文背景环境中同时进行多个主题的开发时，事情只会变得越来越糟糕：



作为一个临时的解决方案，你可以为每一个主题建立一个目录，并且拷贝整个项目到这些目录中去。但是这样又会产生很多其它的问题：

- 你绕过了版本管理系统，因为这些新的目录没有被纳入版本管理系统中。
- 没有了版本控制，你就不能简单地共享你的改动给其它人了，并且不能进行有效的团队协作了。
- 进行完成后的整合（特别是在整合这些改动回到项目中去）就会非常困难，而且非常容易出现错误。

简单来说，如果你想要进行更为专业的操作，你就必须找到一种专业的方法来解决这种多主题并行开发时带来的麻烦。

用分支来解决

你可能已经猜到了吧，利用分支功能就是来解决这些所有问题的最好方法。因为一个分支就代表了一个项目开发过程中的一个主题，并且它们之间是相对独立的。



你在任何时间的所有的更改只适用于你所工作的当前活动（*currently active*）分支，所有其他的分支都不会受到影响。这样一来整个团队就有了很多自由空间来对不同主题进行同步开发，更重要的就是那些带有实验性质的改动，当然你不想为此把整个项目搞砸！如果出了问题，你可以随时重新来过，也可以放弃它或者切换到另一个主题。

非常幸运的是分支功能（branching）在 Git 中被设计得非常简单和方便。当你要开始一个新的主题时，无论这个主题是大是小，你都必须为它创建一个新的分支。

黄金法则

#3: 使用分支功能

分支是 Git 一个非常强大的功能，当然不是偶然的，自始至终， Git 的宗旨就是提供一个即快速又简单的的分支功能。它是一个优秀的工具，并且可以帮助解决开发人员在日常团队开发中存在的代码冲突的问题。因此分支功能应该被广泛地运用在不同的开发主题中。比如添加新功能，修复错误，尝试新的想法等等。

在分支上工作

到目前为止,我们还没有在我们项目上使用过分支。然而你并不知道，我们实际已经工作在了一个分支上了。这是因为在 Git 上的分支功能并不是可选的，你永远会工作在一个分支中的（当前的 “**active**”，或者 “**checked out**”，或者 “**HEAD**”分支）。

那么 Git 是如何知道你当前在哪个分支上工作的呢？ “git status” 命令输出的第一行会向我们显示出 “在主分支（branch master）上”。 这个 “master” 分支是 Git 在建立项目的同时自动为我们建立的。尽管可以删除或者是为它重命名，但是你很少能看到一个没有 “master” 分支的项目，因为基本上大家都会保留它。在这里不要觉得这个 “master” 代表一个很特殊的含义，或者是它是一个与众不同的分支，它其实就是一个和别的分支一样普通的分支而已！

现在让我们来在项目上开始开发一个新的功能吧！在当前的项目状态下，我们建立一个新的分支，并且命名为 “contact-form”：

```
$ git branch contact-form
```

使用 “git branch” 命令来显示出所有在项目中已经存在的分支，而且可以使用参数 “-v” 来显示出很多的信息：

```
$ git branch -v
  contact-form 3de33cc Implement the new login box
* master       3de33cc [ahead 1] Implement the new login box
```

你可以看到那个我们新建立的分支 “contact-form”，而且它是基于相同版本的 “master” 分支。除此之外，那个星号（*）旁边的 “master” 代表了这个分支是我们当前的 HEAD 分支。必须强调一下 "git branch" 命令仅仅“建立”了一个新的分支，但不会“自动切换”到这个分支上去。在我们切换到那个新的分支之前，最好我们使用一个 “git status” 命令来看看当前项目的状态：

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#    directory)
#
#       modified:   imprint.html
#
no changes added to commit (use "git add" and/or "git commit -a")
```

虽然现在仍然还有一些对文件 “imprint.html” 的改动在工作副本（working copy）中，但是现在我们必须马上切换到那个新的 “contact-form” 分支上进行新功能的开发了。但这个改动并不属于这个新定义的功能，我们该怎么做呢？一种方法就是，提交这个还未完成的工作，以后

继续来完成它。但是提交一个还未完成的工作是个很不好的习惯。

黄金法则

#4: 不要提交一个还未完成的工作

不要提交一个还未完成的工作，但这也并不意味着在提交前你必须要完成这个功能定义的所有需求。恰恰相反，对于一个很大的功能模块来说，要把它正确分割成小的完整的逻辑块，用来进行频繁的提交。但是，千万不要为了得到一个干净的工作副本（working copy）而提交一些不完整的改动。在这种情况下，你可以考虑使用 Git 提供的 “Stash” 功能。

暂时保存更改

一个被提交了的改动会被永久地保存在仓库（repository）中。然而，在你日常工作中你经常需要“暂时地”保存一下你的一些本地改动。例如，如果你正在开发一个新的功能，但是与此同时又得到了一个错误报告，并且需要马上修复它，而你现在的本地改动又和这个错误毫无关系，因此你必须暂时地停止新功能的开发，来开始着手修复这个错误。并且你还想要保存那些已完成的开发工作，以便之后能继续来完成它。

像这样的情况会随时发生，比如你必须要开始一个新的工作，而在你的当前工作版本中还有一些并不想立即提交的本地改动。在处理好这些本地改动的同时，我们还需要把当前的工作副本（working copy）清理出来，Git 提供的 “储藏（Stash）” 功能可以非常好地解决这个问题。

概念

储藏（Stash）

可以把储藏想象成一种剪贴板，它会获取你工作副本（working copy）中的所有改动，并且保存到一个新的剪贴板上。然后你就会得到一个“干净”的工作副本，也就是说一个不存在任何改动的工作目录。

之后你随时都可以重新调回那些保存在剪贴板中的改动到你的工作副本中来，从而继续你之前没有完成的工作。

你可以建立多个储藏单元，不仅仅局限于存储一组变化。同样，储藏也会不绑定在你所处的当前分支或是任何其它分支上，如果你想要调回任意一个储藏单元，它的改动将会被应用在你当前的 HEAD 分支上。

来让我们先把本地的改动储藏起来，这样在开始这个新功能开发前我们就可以得到了一个干净的工作副本：

```
$ git stash
Saved working directory and index state WIP on master:
   2dfe283 Implement the new login box
HEAD is now at 2dfe283 Implement the new login box
```

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

现在，这个本地文件 “imprint.html” 的改动已经被安全地保存在一个剪贴板上了。当我们希望继续对它们进行工作时，我们都可以随时轻松地调回它。

你可以很容易地得到你当前的储物箱的状态：

```
$ git stash list
stash@{0}: WIP on master: 2d6e283 Implement the new login box
```

最新建立的储藏单元会被显示在列表的最上面，被命名为 named “stash@{0}” 。早前建立的储藏单元会拥有一个更高的数字。

当你想要调回一个之前建立的储藏单元，有两种方法：

- (a) 使用 “git stash pop” 命令，它将调回最新的一个储藏单元，并且把它从剪贴板中删除掉。
- (b) 使用 “git stash apply <stashname>” 命令，它将调回那个你所给出的储藏单元，而这个储藏单元还会保留在剪贴板中。你可以随时使用 “git stash drop <stashname>” 来删除它。

当你使用这些命令时，你不必给出特定的储藏单元名称。 Git 将会自动地处理最新的那个储藏单元（永远是 “stash@{0}”）。

概念

储藏的时机

储藏功能可以帮助我们得到一个干净的工作副本。当然，它还可以应用在很多不同的流程中，强烈推荐你在下列情况中储藏你的本地改动：

- ……在切换到不同分支之前。
- ……在获取（pulling）远程改动之前。
- ……在合并（merging）或者衍合（rebasing）一个分支之前。

终于，是时候开始着手开始开发我们的新功能了！

切换一个本地分支

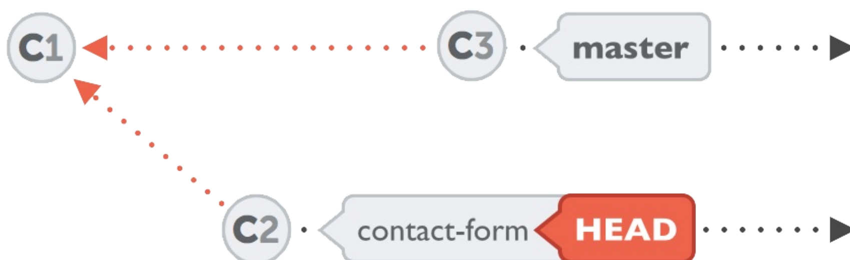
现在我们得到了一个干净的工作副本，第一件事就是要切换到，或者说“签出（check out）”那个新建的分支上去：

```
$ git checkout contact-form
```

概念

签出（Checkout)，HEAD，和你的工作副本（Working Copy)

分支会自动指向最后一次的提交。而且，一个提交也对应项目中的一个特定版本，Git 总是非常地清楚定位哪些文件属于该分行。



在这个时间点，仅仅有一个分支被指向 HEAD，或者说仅仅有一个被签出（checked out）的活动分支。在你的工作副本上的文件都会被关联在这个分支上。所有其它的分支以及它们的关联文件都被安全地保存在 Git 的数据库中了。

指定另外一个分支为活动分支（比如我们之前建立的 “contact-form” ），可以使用 “git checkout” 命令。 这个命令会为我们完成两件事：

- （a） 它会让 HEAD 指针指向这个 “contact-form” 分支。
- （b） 它会替换你工作目录（working directory）中的所有文件，并且完全匹配它们的版本到 “contact-form”。

再执行一下 “git status” 命令，你将看到我们现在已经切换到分支 “contact-form” 上了。从现在开始我们所有的改动和提交都将只适用于这个分支，直到我们再次使用 “checkout” 命令切换到其它分支上去。

让我们来验证一下。建立一个新的文件并且命名为 “contact.html” 然后提交它：

```
$ git add contact.html
$ git commit -m "Add new contact form page"
$ git log
commit 56eddd14cf034f4bcb8dc9cbf847b33309fa5180
Author: Tobias Günther <support@learn-git.com>
Date: Fri Jul 26 10:56:16 2013 +0200

    Add new contact form page

commit 2dfe283e6c81ca48d6edc1574b1f2d4d84ae7f1
Author: Tobias Günther <support@learn-git.com>
Date: Fri Jul 26 10:52:04 2013 +0200

    Implement the new login box

commit 2b504bee4083a20e0ef1e037eea0bd913a4d56b6
Author: Tobias Günther <support@learn-git.com>
Date: Fri Jul 26 10:05:48 2013 +0200

    Change headlines for about and imprint
```

注意观察这个日志信息，你会看到那个新提交的文件被正确保存下来了，到目前为止这并没有什么特别的。但是现在让我们切换回 “master” 分支，并且再来观察下一个它的日志信息：

```
$ git checkout master
$ git log
commit 2dfe283e6c81ca48d6edc1574b1f2d4d84ae7f1
Author: Tobias Günther <support@learn-git.com>
Date: Fri Jul 26 10:52:04 2013 +0200

    Implement the new login box

commit 2b504bee4083a20e0ef1e037eea0bd913a4d56b6
Author: Tobias Günther <support@learn-git.com>
Date: Fri Jul 26 10:05:48 2013 +0200

    Change headlines for about and imprint
```

你会发现到那个注释为 “Add new contact form page” 的提交并不在这里，这是因为我们操作仅仅只针对于当时的那个 HEAD 分支（当时的 HEAD 分支应该是 “contact-form”，而不是 “master")。这正是我们想要的结果，我们的改动应该仅仅保持在它对应的分支环境中，并不会影响其他的分支环境。

合并改动

保持你的提交只在独立分支环境下是很有意义的。但是当你想要将这个提交的改动整合到别的分支中去时，就会需要一些额外的工作。例如，你完成了一个新功能的开发，你想要把这个功能整合到你的“产品”分支中去。或者相反的流程，你正在一个分支中开发这个新的功能，同时在你所开发项目中发生了一些改变（比如一些严重的错误被修复了），你很需要这些改动也能被整合到你正在使用的分支中。

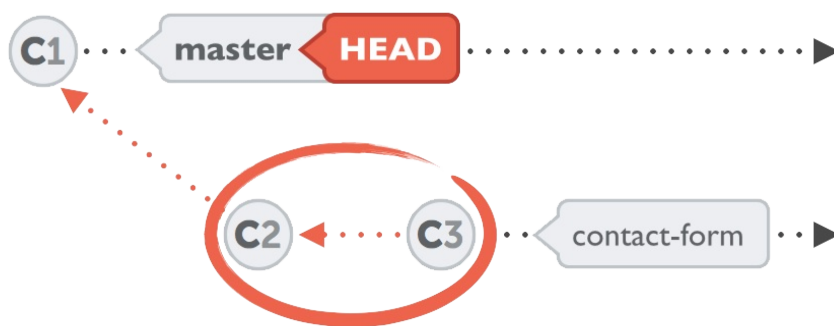
无论是哪一种情况我们都称这种整合叫做“合并（merging）”。在 Git 中我们使用“git merge”命令来进行合并的操作。

概念

整合分支-不是单独提交

在开始准备合并时，你不必（当然也不能）把那些要整合的改动打包为一个单独的提交。相反，你要告诉 Git，你想要和那个分支（*branch*）进行整合，Git 会从那个分支中判断出哪些提交还没有被整合到你当前工作的 HEAD 分支中。只有这些提交才会被整合进来。

此外，你不需要去考虑这些改动最终会到了哪里，整合的目标永远是你的当前的 HEAD 分支，也就是你的工作副本。



在 Git 中，进行合并是非常简单方便的。它只需要两个步骤：

- （1） 切换到那个需要接收改动的分支上。
- （2） 执行“git merge”命令，并且在后面加上那个将要合并进来的分支的名称。

来让我们把“contact-form”分支的改动合并到“master”中去：

```
$ git checkout master
$ git merge contact-form
```

现在如果你执行“git log”命令，你会看到那个提交“Add new contact form page”已经被成功地合并到 master 分支中来了！

```
$ git log
commit 56eddd14cf034f4bcb8dc9cbf847b33309fa5180
Author: Tobias Günther <support@learn-git.com>
Date: Fri Jul 26 10:56:16 2013 +0200

    Add new contact form page

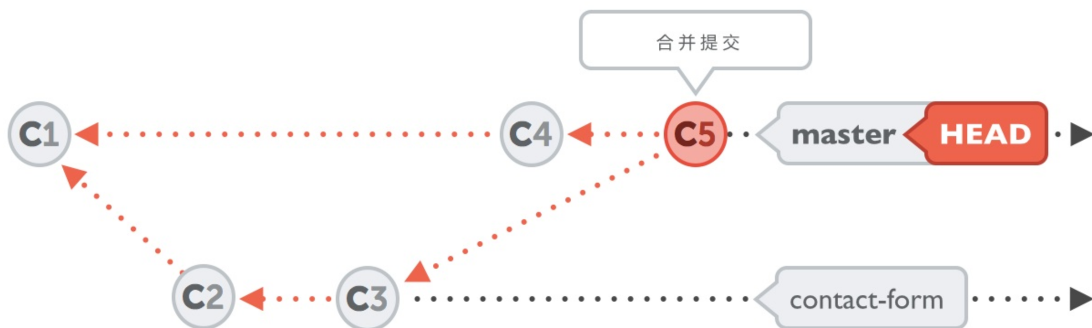
commit 2dfe283e6c81ca48d6edc1574b1f2d4d84ae7f1
Author: Tobias Günther <support@learn-git.com>
Date: Fri Jul 26 10:52:04 2013 +0200

    Implement the new login box

commit 2b504bee4083a20e0ef1e037eea0bd913a4d56b6
Author: Tobias Günther <support@learn-git.com>
Date: Fri Jul 26 10:05:48 2013 +0200

    Change headlines for about and imprint
```

然而，合并操作的结果并不是都能很清楚地被显示出来。 Git 并不是简单地将那些需要的提交整合到你的 HEAD 分支中去，它经常会结合出一个新的改动，然后执行 “merge commit” 进行一次单独的提交。你可以把这种提交想象成连接两个分支的节点。



你可以随时经常性地合并两个分支。每次 Git 都会检查那个将要合并进来的分支上的提交，并且只整合那些还没有合并过的提交。

参考

有时进行合并操作会产生一个或多个 “合并冲突（merge conflicts）”，在这种情况下 Git 就不能自动地连接那些改动。例如，在两个分支中都修改了同一个文件的同一行，这时你要自己来决定哪些改动是你想要最终保留的。我们将在本书之后的章节里会为你介绍这方面的操作 处理合并冲突。

分支的工作流程

分支的工作流程要取决于它的使用背景，我们可以将它分为两个主要的方面。

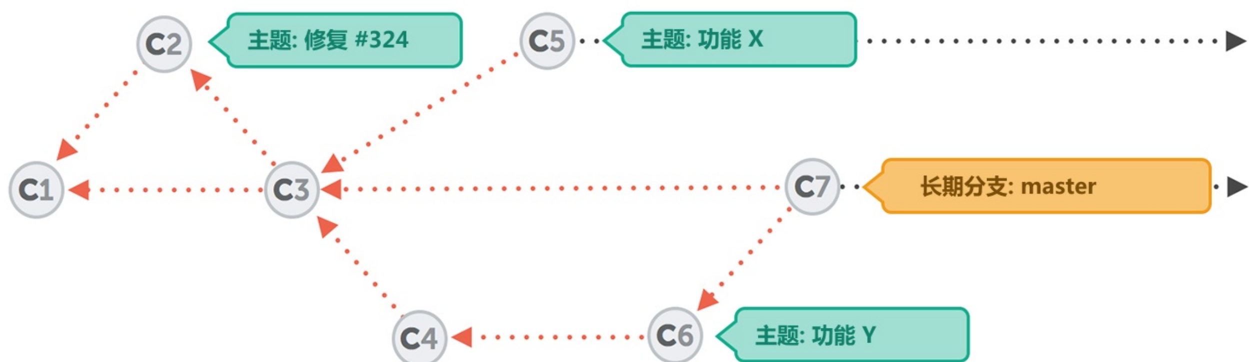
注释

请记住，在这里它只是一个语义层面上的划分。在技术和实用层面上，一个分支就是一个分支，它们的原理都是一样的。

(A) 短期分支（Short-Lived）/主题分支（Topic Branches）

在本书前面的章节中已经提到了我对建立分支的一些建议，例如：对应新功能的分支，修复错误的分支 以及 进行代码尝试所建立的分支。这些分支都有两个重要共同特征：

- 它们只涉及一个单一主题，而且它包含的代码要和它的主题相对应。例如，你不应该建立一个关于购物车功能的分支，并且再在这个分支上去提交一些有关于邮件订阅功能和错误修复的改动。
- 它们都只有非常短暂的生命周期。通常情况下，这个生命周期只维持到这个开发主题的结束之后（例如当错误被修复，新功能被完成......），这个分支的改动就会被整合到项目的大环境中去，并且这个分支也会被随之删除掉。



(B) 长期分支 Long-Running Branches

相对于那些基于一个单一开发主题或是一个错误修复的短期分支来说，这类分支被定义在更高的层面上。它们代表了在整个项目生命周期的一种状态（比如："产品"，"测试"，"研发"状态）。它们在项目中保存的时间比较长，甚至可能是整个项目的生命周期。这类分支有如下一些典型特点：

- 你不应该直接在这个分支上工作。但是你可以整合其他的分支（例如功能分支或是其他

的长期分支）到这类分支中来，尽量不要对它直接进行提交。

- 一般来说，在长期分支之间也存在不同的等级。例如 “master” 分支一般被定义为最高等级。它应该只保存产品代码（production code）。在它之下应该存在一个 “开发（development）” 分支。它会被用来进行真正的开发和测试工作，然后整合入 “master” 分支……

我们应该为项目建立哪些长期分支呢？该如何使用它呢？这并不能一概而论。这在很大程度上取决于开发团队的规模，开发风格和项目的需求，甚至于不同的客户。这个规则必须被很清楚的定义出来，并且要得到整个开发团队的认同和遵守。

一个简单通用的分支策略

我们已经讲到了，不同的开发团队必须定义合适自己的分支策略。然而，还是有一种简单通用的流程，它应该适用于大多数的开发团队。

仅仅使用一个长期分支

虽然你可以在你的开发流程中引入多个长期分支，但是这样也会存在很多不利因素。最为显著的是会让你开发流程变得非常复杂！在你的工作流程中仅仅定义和使用一个单一的长期分支可以避免很多不必要工作，并且使项目的开发变得比较简单。

概念

在一般的情况下，“master” 分支可以有效地代表你的产品代码。然而一个重要的前提就是，所有被合并到 **“master”** 分支的代码都必须保证正确！ 你必须保证它们的质量。因此它们必须经过测试，它们的代码必须被检验过和确认过。

这也意味着，开发工作不应该直接在 “master” 分支上进行，这也是一个最基本的准则。因此当你使用了 “git checkout master” 命令切换到 “master” 分支后并提交改动时，你就要问问自己，这样是否符合流程？这些改动是否能保证正确？

主题分支

每当你开始着手开发一个新的功能的或是修复错误时，你都应该对应不同的主题建立一个新的分支。这是一种很通用的做法，并且也要成为你的一种习惯。

如果在你的项目仓库中只存在一个长期分支，那么所有的主题分支都必须基于这个 “master” 分支。当你所开发的功能完成后，或者是错误被修复后，这些所对应的改动就理所当然的要被合并回 “master” 分支。

在你开发你的新功能的同时，团队的其他开发人员已经把各自完成的改动整合回了 “master” 分支。在这种情况下，你就必须经常性地把那些在 “master” 分支上的改动合并到你的工作分支上来。这就确保了你的工作分支一直处于最新的状态,并且当你要把已完成的改动整合回

“master” 分支上时，可以减少可能出现的冲突和风险。

请不要忘记这个简单的黄金法则：只有正确和稳定的代码才能被整合到 “master” 分支上来！如何确保这些代码正确性呢？这就依靠你和你的团队了。例如使用单元测试（unit tests），代码审查（code reviews）等等。

保持远程同步

在 Git 中，远程和本地仓库可能彼此毫无依赖关系。不管怎样，本地和远程的分支必须被一致对待。

但是这样也并不代表你必须要发布你的 每一个 本地分支，拥有一些完全属于你的私有分支也是非常必要和有意义的。例如当你单独的研究一些新功能，或是尝试调试一些新的技术时。不管怎样，当你要发布你的一个本地分支时，你就应该给它对应的远程分支一个相同的名称。例如，如果你有一个本地分支被命名为 “login-ui”，当你要发布它时，在远程仓库上它也必须拥有 “login-ui” 这个名字。

频繁推送

保持与远程分支的同步并不是只停留在结构层面上，经常性地通过 “git push” 命令发布你的改动可以有助于团队里的其他的开发人员看到和使用你的最新开发成果。附带的还有一个最大的好处是，它可以作为你的远程备份。

其他分支策略

上述策略是最适用于小型的开发团队。而一个较大的开发团队可能会需要更多的规则和更复杂的结构。 在网络上搜索其他开发团队的策略将为你提供更多的选择。这是其中一个非常流行并且可能是值得尝试的一种工作流程 “git-flow”。

注释

我个人认为 git-flow 有些过于强大了：

- 它拥有自己的脚本来诠释 Git 和拥有自己独有的命令。这使得它很难在一个 GUI 应用程序中被使用。
- 在它竭尽全力地简化 Git 的同时， 它又需要让使用者去学习一种几乎是全新语言定义的命令。

通过正确的学习 Git 的基本知识，并且在开发团队中确定一个专属的共同工作流程后，你可能会认为类似 git-flow 所定义的扩展有些过于束缚。

Part 3 - 远程仓库

关于远程仓库

在版本控制系统上，大约90%的操作都是在本地仓库（local repository）中进行的：暂存，提交，查看状态或者历史记录等等。除此之外，如果仅仅只有你一个人在这个项目里工作，你永远没有机会需要设置一个远程仓库（remote repository）。

只有当你需要和你的开发团队共享数据时，设置一个远程仓库才有意义。你可以把它想象成一个“文件管理服务器”，利用这个服务器可以与开发团队的其他成员进行数据交换。

现在让我们来看一下本地仓库与远程仓库之间的区别吧：

位置

本地仓库位于每一个团队成员的本地计算机上。相反，远程仓库则被设置在一个能被所有团队成员访问到的远程服务器上，基本上都是在互联网上或者是本地局域网中。

特点

从技术上讲，一个远程仓库和一个本地仓库并不存在很大差异。和本地仓库一样，它包含分支，提交和标记。然而，本地存储库存在一个与之相关的工作副本（working copy），就是你当前的工作目录。其中包括了你从项目中签出的一些版本文件。而一个远程仓库就不包含任何工作目录，它仅仅是由一系列 “.git” 目录组成的。

创建

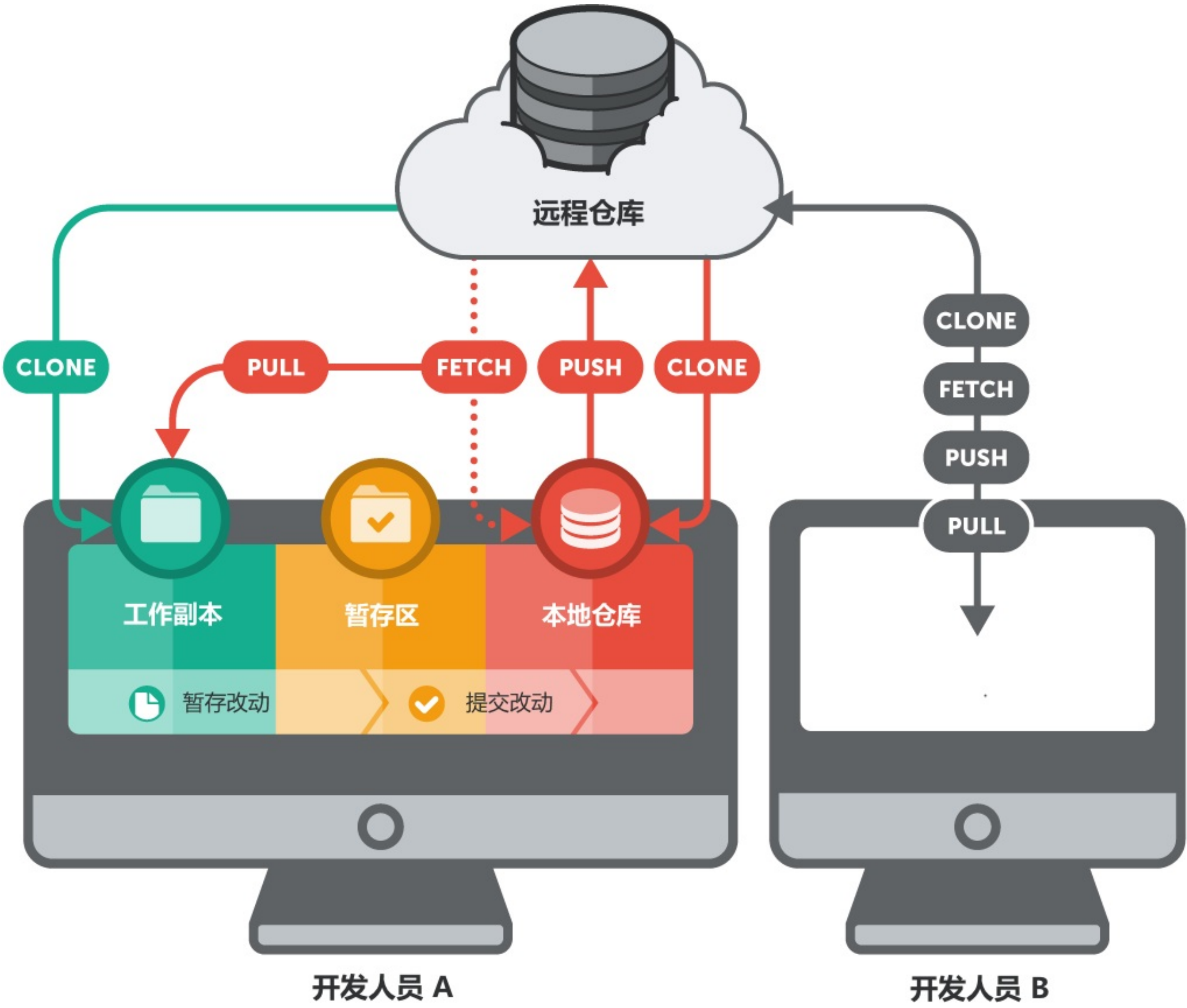
你有两种方法来创建一个本地仓库到你的本地计算机上，你可以新建一个空的仓库，或者是克隆一个已存在的远程仓库到你的本地计算机中。 创建一个新的远程仓库同样的也有两种方法。方法一，假如已经存在了一个本地仓库，并且你想以这个本地仓库作为远程仓库的基准，你使用 “--bare” 参数来可以克隆这个本地仓库。方法二，假如你想要新建一个空的远程仓库，你可以使用 “git init” 命令，并且同时也要加上 “--bare” 参数。

本地 / 远程工作流程

在 Git 中，仅仅存在一小部分应用在远程仓库上的命令。

绝大多数工作都是针对本地存仓库的。到现在为止（除去克隆命令 “git clone” 外），我们一直只是工作在我们本地仓库上，并且从没有离开过本地计算机。我们不依赖于任何互联网或局域网连接，完全是脱机工作的。

我们将会在本章的之后小节中来一起学习一些远程仓库上的命令。


远程仓库
CLONE
PULL
FETCH
PUSH
CLONE
CLONE
FETCH
PUSH
PULL
工作副本
暂存区
本地仓库
暂存改动
提交改动
开发人员 A
开发人员 B

连接一个远程仓库

当你克隆一个远程仓库的同时，Git 会自动为你的记录下它的链接。默认使用这个名字“origin” 来标识你所克隆的原始仓库。 如果你是直接在计算机上创建了一个本地仓库，这样就没有任何一个远程链接被记录下来。这种情况下，当你尝试做任何远程操作之前你就必须先把它连接到一个远程仓库上去：

```
$ git remote add crash-course-remote
    https://github.com/gittower/git-crash-course-remote.git
```

来让我们来看看它的结果：

```
$ git remote -v
crash-course-remote   https://github.com/gittower/git-crash-course-remote.git (fetch)
crash-course-remote   https://github.com/gittower/git-crash-course-remote.git (push)
origin   https://github.com/gittower/git-crash-course (fetch)
origin   https://github.com/gittower/git-crash-course (push)
```

请注意，每个远程仓库包含两行，第一个是用来进行抓取的 “fetch URL”，第二个是用来把本地仓库中的数据推送到远程仓库 “push URL”。很多情况下这两个 URLs 都是相同的。然而你当然也可以对抓取（fetch）和推送（push）使用两个不同的 URLs（例如出于安全和性能方面的考虑）。

此外还要注意到，你可以对一个本地仓库设置很多个远程链接，这是没有数量限制的。在上面的例子中你已经看到了一个已经存在的链接 “origin” ，其实我们从来也没有设置过它！在完成某个远程仓库的克隆之后，Git 会默认的建立一个名为 “origin” 的远程仓库链接（还记得吗？我们曾在本书的开始部分做过这个操作）。和被命名为 “master” 的分支一样的道理，“origin” 这个名字是默认的。它和其他的远程仓库并没有什么区别。

查看远程数据

在我们新建了与某个远程仓库的连接之后，它到底改变了什么？现在让我们来看看分支列表：

```
$ git branch -va
  contact-form           56eddd1 Add new contact form page
* master                 56eddd1 Add new contact form page
  remotes/origin/HEAD    -> origin/master
  remotes/origin/master  2b504be Change headlines for about and imprint
```

显然并没有太大的变化，始终是我们那两个本地分支（“master” 和 “contact-form”）以及两个在 “origin” 上的远程分支（“remotes/origin/HEAD” 和 “remotes/origin/master”）。为什么我们没有看到那个新的远程链接 “crash-course-remote” 呢？因为通过命令 “git remote add”，我们仅仅建立了一种关系，还没有进行任何数据交换。

概念

远程数据是一个快照（**Snapshot**）

Git 会在你的本地仓库中保存远程数据的信息（例如分支，提交等等）。但是它并不是 “实时地” 连接到你的远程仓库上的。例如，其他团队成员在这个远程仓库中所提交的新改动或是发布的分支，是不能自动地与你分享的，因为你必须明确地告诉 Git 去升级你的本地仓库！

关于远程的分支，远程的提交等等信息只会按照你的要求更新到最新的一个快照。Git 不会在后台 “自动” 升级这些信息。

要更新有关远程的信息，你必须明确地请求这个数据。在这里可以使用最为常用的，“抓取（Fetch）” 操作来完成：

```
$ git fetch crash-course-remote
From https://github.com/gittower/git-crash-course-remote
 * [new branch]     faq-content -> crash-course-remote/faq-content
 * [new branch]     master -> crash-course-remote/master
```

“抓取” 操作不会改动你任何的本地分支或是在你工作副本中的文件。这个操作仅仅为你从一个特定远程仓库下载你所需要的数据，并设置为可见。你可以在之后决定是否需要整合这些新的改动本地项目中来。

在我们完成了关于 “crash-course-remote” 升级后，让我们一起来看一下现在发生了什么变化：

```
$ git branch -va
  contact-form           56eddd1 Add new contact form page
* master                 56eddd1 Add new contact form page
  remotes/crash-course-remote/faq-content  e29fb3f Add FAQ questions
  remotes/crash-course-remote/master       2b504be Change headlines f...
  remotes/origin/HEAD    -> origin/master
  remotes/origin/master  2b504be Change headlines for about and imprint
```

很好，我们看到了这个远程分支的 “crash-course-remote” 的信息了。

现在准备开始在分支 “faq-content” 上工作吧！但是现在它只是一个远程分支的指针。为了能够真正地在这个分支上工作，并且切换当前工作副本（working copy）的内容，我们需要创建一个基于这个远程的本地分支。执行 “git checkout” 命令来切换到这个远程分支：

```
$ git checkout --track crash-course-remote/faq-content
Branch faq-content set up to track remote branch faq-content from crash-course-remote.
Switched to a new branch 'faq-content'

$ git branch -va
  contact-form           56eddd1 Add new contact form page
* faq-content            e29fb3f Add FAQ questions
  master                 56eddd1 Add new contact form page
  remotes/crash-course-remote/faq-content  e29fb3f Add FAQ questions
  remotes/crash-course-remote/master       2b504be Change headlines f...
  remotes/origin/HEAD    -> origin/master
  remotes/origin/master  2b504be Change headlines for about and imprint
```

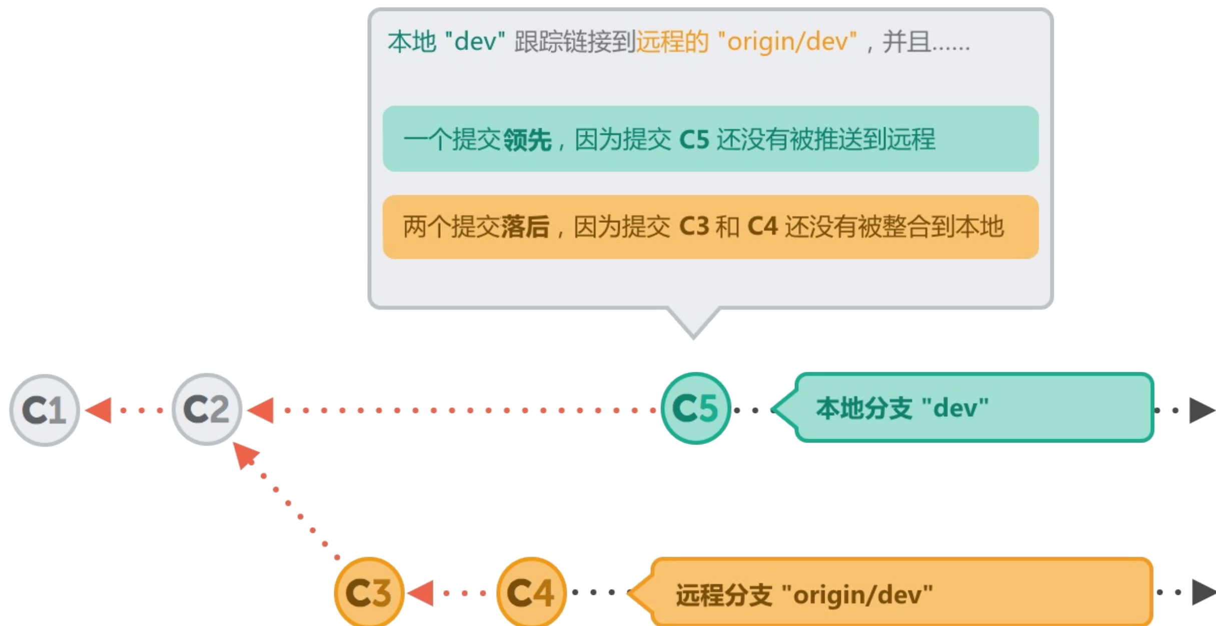
这个命令完成了一系列的操作：

- （a） 它创建了一个同名的本地分支（“faq-content”）。
- （b） 它签出（check out）了这个新建的的分支，把它设置成当前的本地 HEAD ，然后更新了你的工作副本，并且关联到分支文件的最新版本上去。
- （c） 由于我们使用了 “--track” 参数，它会在新的本地分支和它所位于的远程分支之间创建一个跟踪联系 “tracking relationship”。

跟踪分支

一般来说，分支之间并无任何关系。然而我们可以定义一个本地分支去 “跟踪 （track）” 一个远程分支。这样 Git 就会通知你，如果那个被跟踪的远程分支发生了一些新的提交，而它们并不存在于这个关联的本地分支中时：

- 如果在你的本地分支上提交了一些改动，而且你也并没有发布它和推送到远程仓库中。相对于这些提交来说你的本地分支就 “领先（**ahead**）” 于那些它所对应的远程分支。
- 如果团队的其他开发人员提交并且发布了一些改动到远程仓库中，这时远程仓库就拥有了那些你还没有下载到本地仓库的提交。你的本地仓库就 “落后（**behind**）” 于它所关联的远程仓库。



如果分支间存在 “跟踪” 联系，当你使用 “git status” 命令时，Git 显示出所有关联分支上的差异：

```
$ git status
# On branch dev
# Your branch and 'origin/dev' have diverged,
# and have 1 and 2 different commits each, respectively.
#
nothing to commit (working directory clean)
```

当在一个已存在的远程分支的基础上来建立本地分支时，创建这个 “跟踪” 联系是很简单的，可以使用 “git checkout” 命令加 “--track” 参数来完成。

在切换到那个新创建的本地分支 “faq-content” 后， 我们已经自动拥有了这个 “跟踪” 联系。来让我们对这个工作副本中的文件 “faq.html” 进行一些修改吧！（如何更改这个文件就不详细介绍了，发挥你的想象力吧）：

```
$ git status
# On branch faq-content
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#    directory)
#
#       modified:   faq.html
#
no changes added to commit (use "git add" and/or "git commit -a")

$ git add faq.html

$ git commit -m "Add new question"
[faq-content 814927a] Add new question
 1 file changed, 1 insertion(+)
```

现在，是时候把这些新的改动共享给其他开发人员了：

```
$ git push
```

注释

在你的本地计算机上，这个 “git push” 命令将会被远程仓库拒绝掉，因为你现在还没有获得这个远程仓库的修改权限。如果你想要继续的尝试这个操作，我建议你建立一个自己的远程仓库，例如：GitHub 或者 Beanstalk。

“git push” 命令将会把当前 HEAD 分支上所有新的提交上传到它所关联的远程分支上去。

概念

回顾“跟踪”联系

“git push” 命令默认地要求我们为它提供两个信息：

- （a） 你想要推送到哪一个远程仓库上去？
- （b） 你想要推送到那个远程仓库上的哪一个分支上去？

这个完整的命令应该是这样的： `$ git push crash-course-remote faq-content` 我们已经设置了 “跟踪” 联系，也就是说我们已经为那个本地分支定义了一个 “ 远程对应的（remote counterpart）” 分支。在我们使用 “git push” and “git pull” 命令时，我们不需要特别地给出那些参数， Git 会自动地使用这些已经定义好的 “跟踪” 信息。

整合远程的改动

开发团队的其他成员在你们共同的远程仓库上共享了他的改动，在这些改动被整合到你的本地副本之前，你需要首先要检查一下这些改动：

```
$ git fetch origin
$ git log origin/master
```

注释

因为在大部分的项目中你都会拥有一个名为 “origin” 的远程链接，所以我们也将在本书中使用这个远程链接的名称作为例子。

现在使用 “git log” 命令来查看最近在 “origin” 的 “master” 分支上所有改动。 如果你决定要整合这些改动到你的本地副本中来， 你就可以使用 “git pull” 命名来完成这个操作：

```
$ git pull
```

这个命令将会从远程分支下载所有的新的提交到你的本地副本中来。它实际上就是一个 “抓取（fetch）” 命令（下载数据） 和 一个 “ 合并（merge）” 命令（整合那些下载的数据到你的本地副本）的组合。

和 “git push” 命令一样，如果你本地的 HEAD 分支还没有创建任何一个 “跟踪” 链接，你就必须告诉 Git，你要从哪一个远程仓库上的哪一分支中抓取数据（例如 “git pull origin master”）。如果已经存在了一个链接，只需要简单键入 “git pull” 就足够了.

整合的目标并不基于存在什么样的跟踪链接，它总是会被整合到你的本地 HEAD 分支中，也就是你的工作副本。

发布一个本地分支

在你明确地决定将一个本地分支发布到远程仓库之前，这些在你本地计算机上创建的分支是不能被其他的团队成员看到的，它只是你的私有分支。这就意味着，你可以保留某些改动仅仅在你私有的本地分支上，而与其他团队成员分享一些其它分支上的改动。

现在让我们来分享 “contact-form” 分支（它直到现在还仅仅是个私有的本地分支）到 “origin” 远程上：

```
$ git checkout contact-form
Switched to branch 'contact-form'

$ git push -u origin contact-form
Counting objects: 36, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (31/31), done.
Writing objects: 100% (36/36), 90.67 KiB, done.
Total 36 (delta 12), reused 0 (delta 0)
Unpacking objects: 100% (36/36), done.
To file://Users/tobidobi/Desktop/GitCrashkurs/remote-test.git
 * [new branch]    contact-form -> contact-form
Branch contact-form set up to track remote branch contact-form from origin.
```

这个命令会告诉 Git 来发布你当前本地的 HEAD 分支到 “origin”上，并命名为 “contact-form”（保持相同的分支名在本地和其对应的远程分支是非常有必要的）。 这个 “-u” 参数会自动地在你本地的 “contact-form” 分支和新建的远程分支之间创建一个 “跟踪” 链接。执行 “git branch” 命令来显示分支信息，并且附带上一些特定的参数，在方括号中就会显示出这个建立的跟踪联系：

```
$ git branch -vva
* contact-form           56eddd1 [origin/contact-form] Add new contact..
  faq-content            814927a [crash-course-remote/faq-content: ahead
                                   1] Add new question
  master                 2dfe283 Implement the new login box
  remotes/crash-course-remote/faq-content e29fb3f Add FAQ questions
  remotes/crash-course-remote/master      2b504be Change headlines f...
  remotes/origin/contact-form             56eddd1 Add new contact fo...
  remotes/origin/master  56eddd1 Add new contact form page
```

当创建了这个新的远程分支后，发布新的本地提交将会非常简单，执行 “git push” 命令就完成这个操作。

如果某个开发人员拥有对这个远程仓库的操作权限，而且他想在这个你发布的 “contact-form” 上工作，他可以在自己的本地计算机上新建一个本地分支，并跟踪到这个远程分支上。这样他也就同样可以提交自己的改动到 “contact-form” 上了。

删除分支

假设我们在 “contact-form” 分支上的工作已经完成了。并且我们也已经把最终的改动整合到了 “master” 分支。现在我们就不再需要这个分支了。把它删除掉吧：

```
$ git branch -d contact-form
```

为了保持一致，我们也有必要删除它所对应的远程分支。附加上一个 “-r” 参数就可以了：

```
$ git branch -dr origin/contact-form
```

Part 4 - 高级应用

撤销操作

撤销是 Git 提供的一个非常优秀的功能，它可以允许你撤消刚刚所做的操作。这就意味着你不必害怕搞砸你正在工作的项目： Git 一直会让你的项目处于一个安全的状态。

修改最后一次提交

无论你如何精心地推敲你的提交，你总是有可能出错的。比如忘记了把一个改动过的文件添加到提交中，或者是输入了错误的提交注释等等。当你认为提交有问题时，你都可以使用 “git commit” 命令，并附带上 “--amend” 参数，这个操作可以非常轻松地来修改你的最后一次提交。 如果你仅仅是想修改上一次的提交注释，你并不需要操作暂存区，简单地再次输入 “git commit --amend” 并附带上正确的注释就可以了：

```
$ git commit --amend -m "This is the correct message"
```

如果你想要添加更多的改动到上一次提交中，你可以像平常一样把这些新的改动添加到暂存区。然后再次使用 “--amend” 参数进行提交：

```
$ git add some/changed/file.ext
$ git commit --amend -m "commit message"
```

黄金法则

#5: 不要修改已经被发布的提交

“amend” 操作是一个非常强大的小帮手，这点你会很快地领会到。但是在你使用它的同时，你一定要考虑到以下一些方面：

- (a) 你只能使用它来修正你的上一次提交。更早的提交是不能使用 “amend” 来进行操作的。
- (b) 你不要对一个已经在远程仓库上被发布，或者说已经被共享的提交进行 “amend” 操作！这是因为 “amend” 操作实际上在后台打包了一个全新的提交来替换旧的提交。如果在这个远程仓库里仅仅只有你一个人在工作，那么这种操作是没有问题的。但是在团队工作中，如果开发团队的其他人员已经得到了你所发布的改动，并且在此基础之上进行了他自己的改动，再次整合一个被修改过的（amended）提交就会出现很多麻烦。

撤销本地改动

当改动还没有被提交之前，它们仍然被称之为 “本地” 改动。这些在你的工作目录（working directory）的修改还仍然在本地，它们属于未被提交的改动（uncommitted changes）。 有时候你对代码进行了一些修改，但是发现这些改动带来的问题比之前还要多。在这种情况下，你可能想要放弃你刚刚的改动，让代码恢复到你改动之前的版本，也就是上次提交之后的状态。

恢复一个文件到上次提交之后的状态，你可以使用 “git checkout” 命令：

```
$ git checkout -- file/to/restore.ext
```

我们已经知道了 “checkout” 命令主要是用来切换分支用的。但是你同样可以给这个命令附带上 “--” 参数，并加上用一个空格来分隔的文件路径。这个操作将撤销在特定文件上所有的未提交的改动。

如果你想要放弃你在工作副本（working copy）中的所有本地改动，并让你的本地副本恢复到上次提交之后的版本，你可以使用 “git reset” 命令：

```
$ git reset --hard HEAD
```

上面这个操作会通知 Git 将你本地副本上的所有文件替换到和 “HEAD” 分支一致的版本(也就是上次提交之后的版本状态)上，并放弃所有的本地改动。

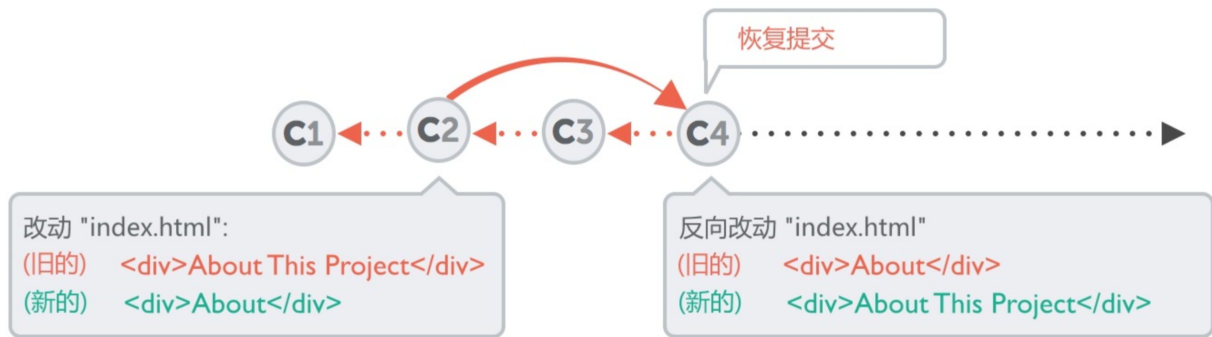
注释

放弃本地未被提交的改动是不能被撤销的。这是因为这些改动还没有保存在你的仓库中。因此，Git 也就没有任何机会来挽回这种操作带来的改动。 请在你撤销本地改动时始终牢记这一点。

撤销已提交的改动

有时你也许想撤销某一个之前的提交。例如当你发现你的改动存在错误，或是整个改动就是错误的，又或者你的客户决定不需要这个改动了等等。

使用 “git revert” 命令可以撤销某个之前的提交。但是这个命令并不是删除那个提交，相反的，它是恢复那个提交的改动，这只是看起来像是撤销而已。这个操作实际上会自动产生一个新的提交。在提交中包括了你想要撤销的那个提交的所有反向改动。例如在原始提交中，你在某一个位置添加一些字符，那么这个恢复提交（reverting commit）就会把这些字符删除掉。



如果想要撤销已提交的改动，你只需要简单地给出这个提交的 commit hash：

```
$ git revert 2b504be
[master 364d412] Revert "Change headlines for about and imprint"
 2 files changed, 2 insertions(+), 2 deletions (-)

$ git log
commit 364d412a25ddce997ce76230598aaa7b9759f434
Author: Tobias Günther <support@learn-git.com>
Date: Tue Aug 6 10:23:57 2013 +0200

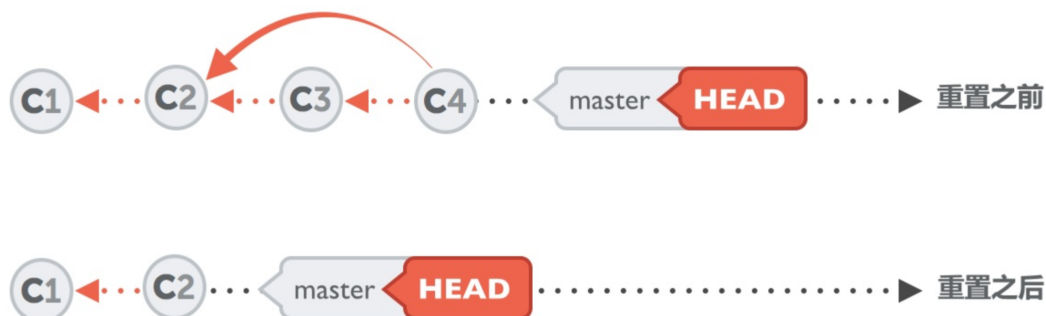
    Revert "Change headlines for about and imprint"

    This reverts commit 2b504bee4083a20e0ef1e037eea0bd913a4d56b6.
```

另外一种撤销提交的方法是使用 “git reset” 命令。这个操作不会自动产生一个新的提交，或是删除你要撤销的提交，它会重置你当前的 HEAD 分支到一个特定旧的版本，也被称作 “回滚（rolling back）” 到旧的版本：

```
$ git reset --hard 2be18d9
```

在执行了这个操作之后，你当前签出的分支将被重置为版本 2be18d9。在这个版本之后的一个或者多个版本将被真正的放弃，它们也不会显示在分支的历史记录中。



如果在这个命令上使用 “--hard” 参数则一定要小心，Git 将会丢弃所有你当前可能拥有的本地改动。整个项目将会被恢复成一个之前的旧版本。 如果你使用 “--keep” 参数来替代 “--hard” 参数，那么在 “回滚” 到的版本之后的所有改动将会转换成本地改动，并保留在你的工作目录中。

注释

和 “revert” 命令一样，“reset” 命令也不会删除任何已存在的提交。这些操作仅仅是做得好像这个提交不存在似的，并从历史记录中删除它们。无论如何，提交会被保存在 Git 的数据库中至少30天。因此，如果你发现你曾经不小心删除了一个仍然有用的提交，任何一个精通 Git 的同事都能为你恢复它们。

这两个命令（revert and reset）只是工作于当前 HEAD 分支上。因此在你执行这些操作之前，你必须要切换到正确的分支上去。

用 diff 来检查改动

项目的开发是由无数个微小的改动组成的。了解项目开发过程的关键就是要搞清楚每一个改动。 当然你可以使用 “git status” 命令或更简单的 “git log” 命令来打印出项目的状态和历史记录，但是这些命令仅仅只能为你提供一个非常简单的信息概要，想要显示一个详细的修改信息就必须使用另外一个命令。

读懂 Diffs

在版本控制系统中用来显示两个版本之间差别的操作我们称之为 “diff”，或者 “patch”。现在就让我们来详细地学习一下这个操作吧！首先要学习如何读懂 diff 信息。

比较的本件 a/b
元数据
a/b 文件标识
区块头
改动 (1)
区块头
改动 (2)
区块头
改动 (3)
区块
区块
区块

```
diff --git a/activesupport/test/test_cases.rb b/activesupport/test/test_cases.rb
index bbeb710..9b62295 100644
--- a/activesupport/test/constantize_test_cases.rb
+++ b/activesupport/test/constantize_test_cases.rb
@@ -34,6 +34,8 @@ module ConstantizeTestCases
     assert_equal Case::Dice, yield("Object::Case::Dice")
     assert_equal ConstantizeTestCases, yield("ConstantizeTestCases")
     assert_equal ConstantizeTestCases, yield("::ConstantizeTestCases")
+    assert_equal Object, yield("")
+    assert_equal Object, yield("::")
     assert_raises(NameError) { yield("UnknownClass") }
     assert_raises(NameError) { yield("UnknownClass::Ace") }
     assert_raises(NameError) { yield("UnknownClass::Ace::Base") }
@@ -43,8 +45,6 @@ module ConstantizeTestCases
     assert_raises(NameError) { yield("Ace::Base::ConstantizeTestCases") }
     assert_raises(NameError) { yield("Ace::Gas::Base") }
     assert_raises(NameError) { yield("Ace::Gas::ConstantizeTestCases") }
-    assert_raises(NameError) { yield("") }
-    assert_raises(NameError) { yield("::") }
   end

   def run_safe_constantize_tests_on
@@ -58,8 +58,8 @@ module ConstantizeTestCases
     assert_equal Case::Dice, yield("Object::Case::Dice")
     assert_equal ConstantizeTestCases, yield("ConstantizeTestCases")
     assert_equal ConstantizeTestCases, yield("::ConstantizeTestCases")
-    assert_nil yield("")
-    assert_nil yield("::")
+    assert_equal Object, yield("")
+    assert_equal Object, yield("::")
     assert_nil yield("UnknownClass")
     assert_nil yield("UnknownClass::Ace")
     assert_nil yield("UnknownClass::Ace::Base")
```

比较文件 a/b

这个 diff 操作会对两个对象进行相互比较。比如对象 A 和对象 B 。在大多数情况下 A 和 B 会是项目中的同一个文件，但它们是基于不同的版本。当然 diff 操作也可以比较两个完全没有关联的文件，并显示出它们之间的差别，但是这种操作并不会被经常使用到。 为了清楚地显示比较信息， diff 操作总是会把要比较的文件定义成 “A” 和 “B”。

文件的元数据 （Metadata）

这所说的文件元数据是非常技术性的，在实践中你可能永远不需要搞明白它。最开始的两串数字表示两个文件的 hashes（简单点说就是它们的 “ID”）。不仅仅是整个项目，Git 还会把每一个文件当作对象来保存。这个 hash ID 就代表了一个文件对象的特定版本。最后的一串数字代表了一个文件的模式（100644 代表它是一个普通的文件，100755 表示一个可执行文件，120000 仅仅是一符号链接）。

标记 a/b

继续向下观察这些输出信息，A 与 B 的真正差别会被显示在这里。为了区分它们，A 和 B 都被赋予了它们特有的符号：对于版本 A，它的符号是一个减号（“-”）；而对于版本 B ，它会使用一个加号（“+”）。

区块（Chunk）

diff 操作不会显示完整的文件内容。如果两个版本仅仅存在两行代码的差别，你也不会想要去逐行地审视这个拥有上万行代码的文件吧。因此，Git 在这里只会标记出那些实际上修改的部分，在这里一段连续的改动被称之为区块（chunk 或者 hunk）。除了包括实际更改的代码行，一个区块还包括一个特定的 “上下文环境”，例如那些改变之前和之后的差别，能让你更容易地明白在特定的上下文环境中这个改变的具体含义。

区块标头（Chunk Header）

每个这样的区块都有一个标头，它被显示在两个 “@@” 符号中。在这里 Git 会告诉你哪些行存在差异。在我们的例子里这些行被标记成为第一个改动区块：

- 来自文件 A （标记为 “-”），从第34行开始之后的6行代码。
- 来自文件 B （标记为 “+”），从第34行开始之后的8行代码。

在那个 “@@” 结束符号之后的信息是用来表明上下文环境的，例如 Git 会尝试着为这个区块赋予一个方法名称或是其他的上下文信息。然而 Git 不能支持所有的文件内容，这很大程度上都要取决于项目所使用的开发语言。

改动

在每一个被改动过的代码行之前都会前置一个 “+” 或是 “-” 符号。就像前面所讲到的，这些符号可以帮助你准确了解版本 A 和 版本 B 。例如前置了 “-” 符号的行就代表来自版本 A，反之带有符号 “+” 的行就代表来自于版本 B。 大多数情况下，在 Git 中都使用 A 和 B 这样的方式，你可以认为 A/- 代表老的内容，而 B/+ 代表新的内容。

现在就让我们来看一下我们的例子：

- 改动 #1 包括两行 “+” ，而在相对应的版本 A 中却不存在这些行（没有任何被前置 “-” 的

行），这就表示这两行是新被添加的。

- 改动 #2 则恰恰相反。在版本 A 中，可以看到有两行被前置上了符号 “-”。然而版本 B 却不存在对应的行（没有 “+” 行），这就表明这两行被删除了。
- 在改动 #3 中，这些代码行发生了一些改动，前置上符号 “-” 的两行被修改了，新的改动就是在它的下面被标记了符号 “+” 的内容。

现在我们知道了如何读懂 diff 的输出信息了，来做一些练习吧！

检查本地改动

在之前的章节里，我们经常会使用 “git status” 命令来查看在本地副本（working copy）中有哪些文件被改动了。如果要想清楚地了解这些改动的细节，我们就必须使用 “ git diff” 命令：

```
$ git diff
diff --git a/about.html b/about.html
index d09ab79..0c20c33 100644
--- a/about.html
+++ b/about.html
@@ -19,7 +19,7 @@
   </div>

   <div id="headerContainer">
-    <h1>About</h1>
+    <h1>About This Project</h1>
   </div>

   <div id="contentContainer">
diff --git a/imprint.html b/imprint.html
index 1932d95..d34d56a 100644
--- a/imprint.html
+++ b/imprint.html
@@ -19,7 +19,7 @@
   </div>

   <div id="headerContainer">
-    <h1>Imprint</h1>
+    <h1>Imprint / Disclaimer</h1>
   </div>

   <div id="contentContainer">
```

在不带任何参数的情况下，“git diff” 会为我们给所有在本地副本中还未被打包（unstaged）的变化做个比较，并显示出来。 如果你仅仅是想要查看那些对于已被打包的改动的比较结果，你可以选择使用 “git diff --staged” 命令。

检查已提交的改动

你已经学习过了 “git log” 命令，它会打印出那些最新提交的概要。但是它仅仅显示一些最基础的信息（hash，作者，时间，注释）。如果你想要查看那些改动的细节，你就可以加上 “-p” 参数来得到一个更详细的修改信息。

比较分支和版本

最后，你可能想要知道如何比较两个分支，或是两个特定项目版本。来让我们看看在 “contact-form” 分支的哪些改动并不存在于 “master” 上：

```
$ git diff master..contact-form
```

相反，这些比较信息仅仅是在分支层面上的，你也可以比较任意的两个项目版本之间的内容：

```
$ git diff 0023cdd..fcd6199
```

处理合并冲突

对于很多人来说，合并时出现冲突是非常可怕的事，这就好像一不小心格式化了自己的硬盘一样。在这一章节里我将为你消除这种恐惧。

你不会把事情搞砸

首先你应该记住，你总是可以撤销一个合并操作，并且返回到冲突发生之前的状态。也就是说，你永远有机会放弃并重新开始。

如果你已经掌握了一些关于其它的版本控制系统的使用经验，例如 Subversion ，你可能会很难过。因为在 Subversion 中处理冲突是被大家公认极为复杂而繁琐的。这也就是为什么我们要使用 Git 的原因。简单地说，它在这方面的工作原理是完全不同于 Subversion 的。Git 能够在合并过程中顾及到很多方方面面的东西，从而为你创造一个比较简单的方案来解决可能出现的冲突。

当然，冲突只会妨碍你自己的工作，它是不会涉及到整个团队的项目仓库。这是因为在 Git 中，冲突只可能发生在开发人员的本地计算机上，而不是在远程服务器上。

什么是一个合并冲突

在 Git 中，“合并（merging）” 是在形式上整合别的分支到你当前的工作分支的操作。你需要得到在另外一个上下文背景下的改动（这就也就是我们所提到过的，一个有效的分支应该是建立在一个上下文工作背景上的），并且合并它们到你的当前的工作文件中来。

作为你的版本管理系统，Git 所带来的最伟大的改善就是它让合并操作变得非常轻松简单。在大多数情况下，Git 会自己弄清楚该如何整合这些新来的变化。

当然，也存在极少数的情况，你必须自己手动地告诉 Git 该怎么做。最为常见的就是大家都改动了同一个文件。即便在这种情况下，Git 还是有可能自动地发现并解决掉这些冲突。但是，如果两个人同时更改了同一个文件的同一行代码，或者一个人改动了那些被另一个人删除了的代码，Git 就不能简单地确定到底谁的改动才是正确的。这时 Git 会把这些地方标记为一个冲突，你必须首先解决掉这些冲突，然后再继续你的工作。

如何解决合并冲突

当面对一个合并冲突时，我们首先要搞明白发生了什么。例如是不是你和你的同事都同时编辑了同一个文件的同一行代码呢？是不是他删除了一个你正在编辑的文件呢？是不是你们同时添加了一个相同文件名的文件呢？ 当你使用 “git status” 时， Git 会告诉你存在一个 “未合并

的路径（unmerged paths）"，这只是用另外一个方式告诉你，存在一个或多个冲突：

```
$ git status
# On branch contact-form
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#       both modified:   contact.html
#
no changes added to commit (use "git add" and/or "git commit -a")
```

就让我们来深入地探讨一下，如何去解决这些最常见的冲突。 当两个改动发生在同一个文件的同一些行上，我们就要看看发生冲突的文件的内容了。Git 会非常友好地把文件中那些有问题的区域在 “<<<<<<< HEAD” 和 “>>>>>>> [other/branch/name]” 之间标记出来。

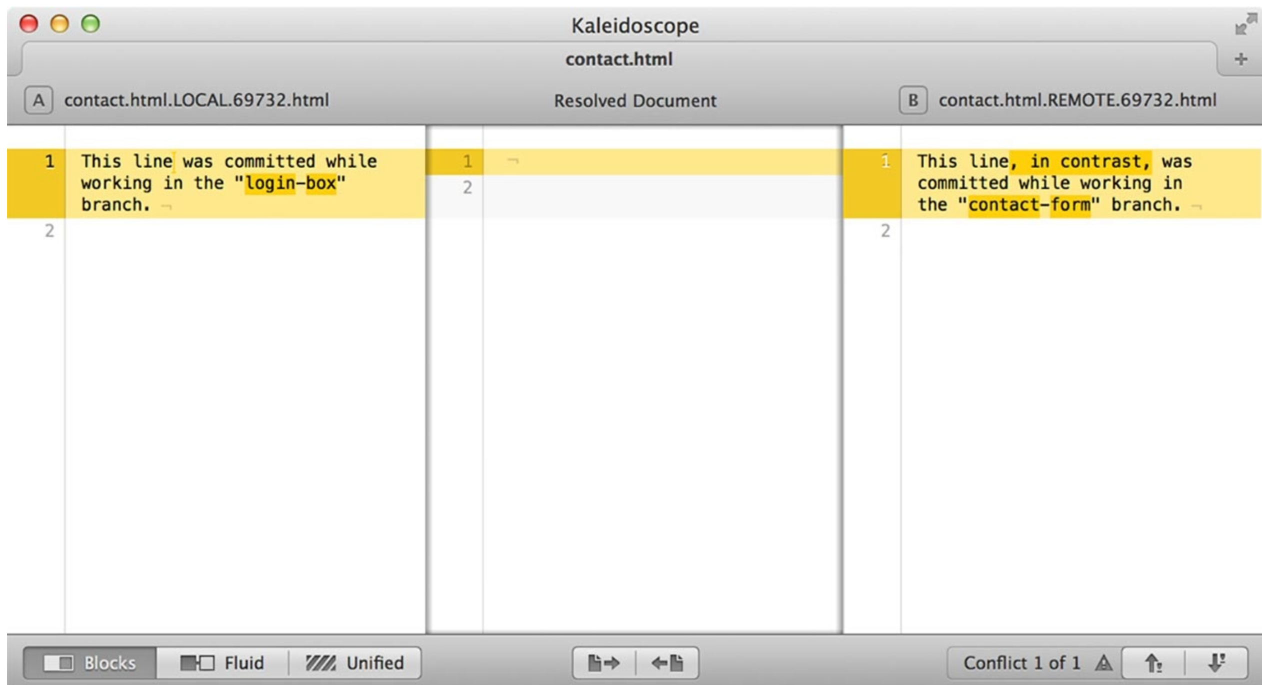
```
1  <<<<<<< HEAD
2  This line was committed while working in the "login-box" branch.
3  =======
4  This line, in contrast, was committed while working in the "contact-form" branch.
5  >>>>>>> refs/heads/contact-form
```

第一个标记后的内容源于当前分支。在尖括号之后，Git 会告诉我们这些改动是从哪里（哪个分支）来的。然后有冲突的改动会被 “=======” 分割起来。

现在我们的工作是要清理这些问题行。当我们完成这些清理后，这个文件应该看起来和我们预期的完全一样。在过程中你也可能需要咨询一下那个和你的代码发生冲突的同事，从而更好地决定哪些改动才是最终正确的，哪些改动是需要被放弃掉的。可能是你的改动，也可能是他的，或者可能是你们两个改动的组合。

打开一个比较原始的文件编辑器来清理这些冲突看起来是可行的，但是这样并不简单。使用一个专门的合并工具可以使这个操作变得更容易（如果你已经安装了一个在你的本地计算机上……）。你可以通过 “git config” 命令来设置这个合并工具给 Git。更详细的内容你就要查看这个工具的文档说明了。 之后当发生合并冲突时，你可以使用 “git mergetool” 命令来调用这个工具。

例如，我在 Mac 上使用 “Kaleidoscope.app”：



在左边和右边的窗口会标记出那些改动的冲突。比起那些用符号 “<<<<<<<” 和 “>>>>>>>” 来标记冲突的方法来说，这是一个更加优雅的可视化环境。你可以非常方便地选择哪个改动是需要被保留的。位于中间的窗口会显示出处理后的结果，并且你也可以进一步手动编辑它。

现在，当清理文件并得到最终代码后，所有剩下的工作就是将这个结果保存起来，并且马上退出这个合并工具。这样 Git 就会知道你已经完成了这个操作。Git 会在后台对那个文件自动地执行 “git add” 命令。这也标志着冲突已经解决了。如果你不使用合并工具，而是手动在文本编辑器中清理这些冲突，你必须手动地将文件标记为已解决状态（通过执行命令 “git add <filename>”）。

最终，当所有的冲突被解决后，你必须通过一个正常的提交操作来完成这个清理合并冲突的工作。

如何撤销一个合并

你应该始终牢记，你可以在任何时间执行撤销操作，并返回到你开始合并之前的状态。要对自己有信心，你不会破坏项目中的任何东西。只要在命令行界面中键入 “git merge --abort” 命令，你的合并操作就会被安全的撤销。

当你解决完冲突，并且在合并完成后发现一个错误，你仍然还是有机会来简单地撤销它。你只须要键入 “git reset --hard <commit-hash>” 命令，系统就会回滚到那个合并开始前的状态，然后重新开始吧！</commit-hash>

Rebase 代替合并

虽然合并（merge）操作可以用来简单和方便地整合改动，但是它却不是唯一的方法。“Rebase” 就是另一种替代手段。

注释

虽然 rebase 相对于我们已知的整合操作来说有着比较显著的优点，但是这也是在很大程度上取决于个人的喜好。一些团队喜欢使用 rebase，而另一些可能倾向于使用合并。

Rebase 相对于合并来说是比较复杂的。我建议你可以跳过这一章，除非你和你的团队确定会用到 rebase 操作。当你积累了 Git 的一些基本使用流程的实践经验后，你也可以在以后的时间再回来学习本章的内容。

深入了解合并操作

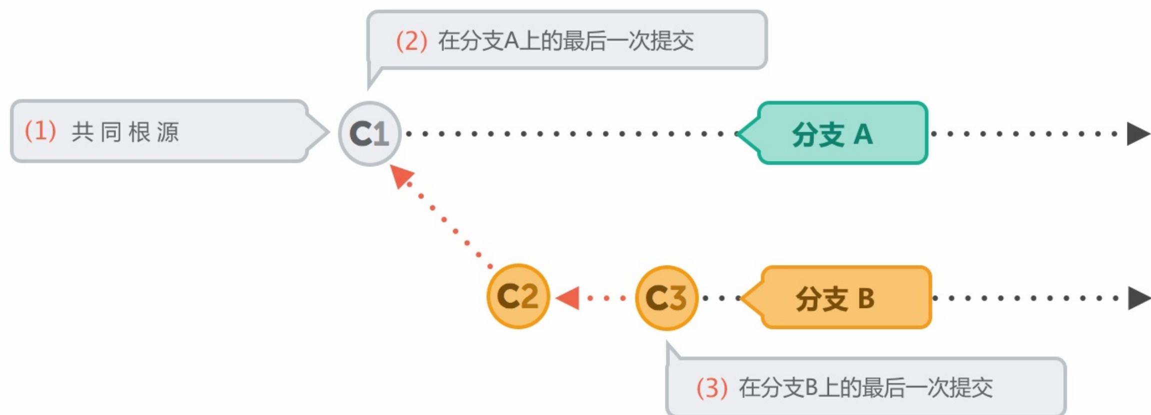
在你进入 rebase 这个主题前，我们有必要来再次探讨一下更多关于合并操作的细节。当 Git 执行一个合并时，它实际上会查找三个提交：

- (1)共同的原始提交 如果你在项目中查看两个分支的历史，它们总是会出自于一次共同的提交，那么在当时的时间点上，这两个分支还是拥有相同的内容。之后它们就开始有了差别。
- (2) + (3) 两个分支的最终点 合并操作的目的就是把两个分支的最新状态结合起来。因此他们各自的最新版本是有特殊含义的。

结合这三个提交后得到的结果就是我们整合的目标。

快进或合并提交

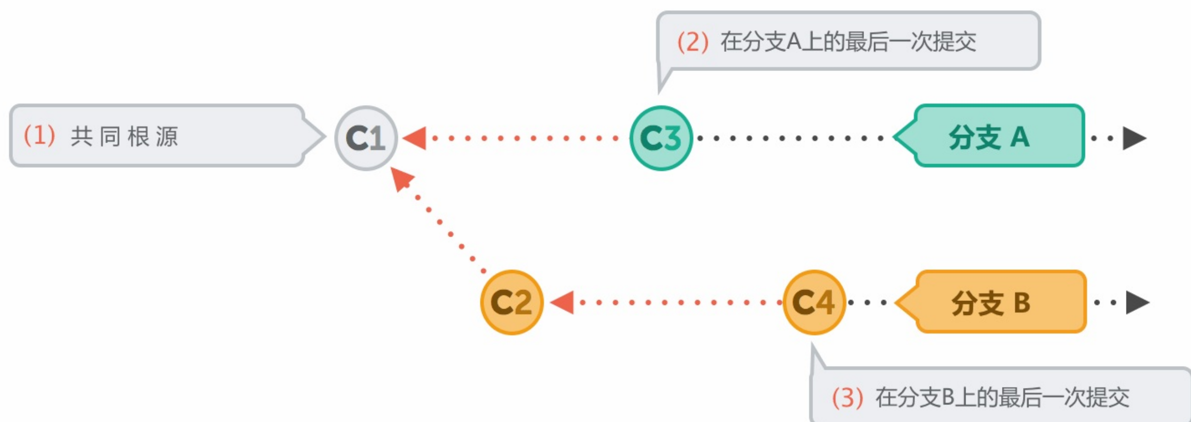
一种最简单的情况是，在其中的一个分支上没有任何一个新的改动提交发生。那么在它之前的最后一次提交就仍然还是那个共同的原始提交。



在这种情况下，执行整合操作就非常简单了。 Git 仅仅需要添加所有那些在另外一个分支上的新提交就可以了。在 Git 中，这种最简单的整合操作我们称之为“快进（fast-forward）”合并。之后两个分支就拥有了完全相同的历史。



但是在大多数情况下，两个分支都会有自己不同的发展轨迹。



为了完成整合，Git 会需要创建一个新的提交来含括它们之间的差异，这就是整合提交（merge commit）。



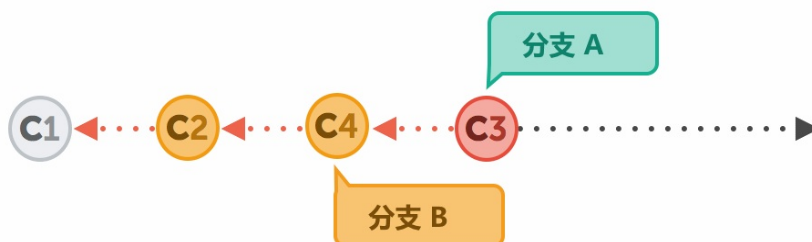
手工提交与合并提交

通常情况下，提交都是由手工精心创建的。这样也就能更好地保证一次提交只涉及一个关联改动，并且能更好地注释这个提交。

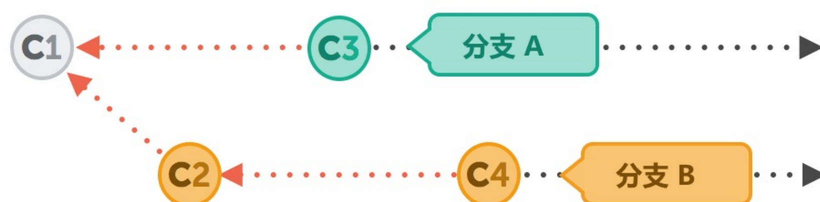
一个合并提交就不同了，它不是由开发人员手动创建的，而是由 Git 自动生成的。它也不涉及一个关联改动，其目的只是连接两个分支，就像节点一样。如果之后想要了解某个合并操作，你只需要查看这两个分支的历史记录和它们相应的提交树（version tree）。

Rebase 整合

有些人并不喜欢使用这种自动合并提交。相反，他们希望项目拥有一个单一的历史发展轨迹。比如一条直线。在历史纪录上没有迹象表明在某些时间它被分成过多个分支。



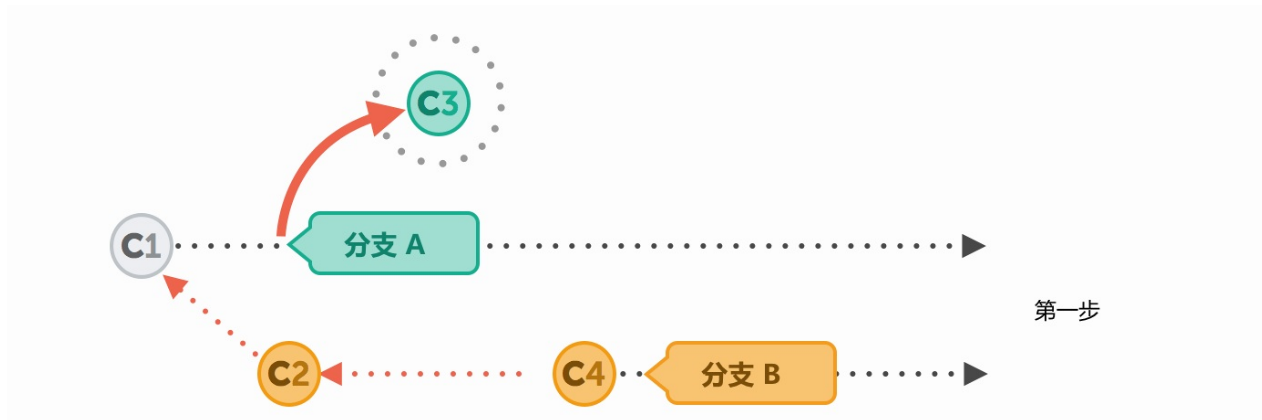
现在就让我们一步一步地了解一下 rebase 操作吧！仍然来使用前面的例子：我们想合并分支 B 到 分支 A 中，但是这次使用 rebase 操作。



使用下面这个非常的简单的命令：

```
$ git rebase branch-B
```

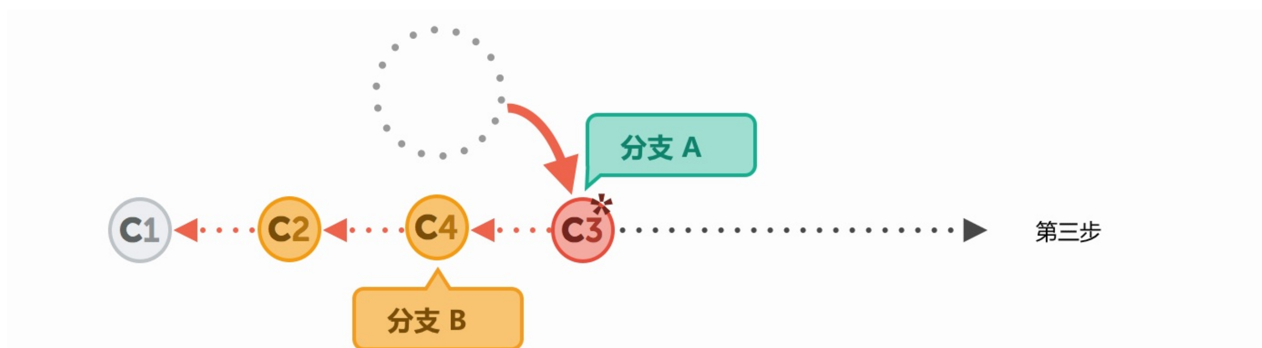
首先，Git 会 "撤销" 所有在分支 A 上的那些在与分支 B 的共同提交之后发生的提交。当然，Git 不会真的放弃这些提交，其实你可以把这些撤销的提交想像成 "被暂时地存储" 到另外的一个地方去了。



接下来它会整合那些在分支 B（这个我们想要整合的分支）上的还未整合的提交到分支 A 中。在这个时间点，这两个分支看起来会是一模一样的。



最后，那些在分支 A 的新的提交（也就是第一步中自动撤销掉的那些提交）会被重新应用到这个分支上，但是在不同的位置上，在那些从分支 B 被整合过来的提交之后，它们就被 *re-based* 了。 整个项目开发轨迹看起来就像发生在一条直线上。相对于一个合并提交，rebase 包括了所有的组合变化，最原始的提交结构会被保留下来。



Rebase 存在的陷阱

当然，使用 rebase 操作不会是永远一帆风顺的。很有可能会搬起石头砸自己的脚，因此你不能忽视一个重要的事实：**rebase** 会改写历史记录。

你有可能已经注意到了，在被 rebase 操作之后的版本中，提交 “C3*” 存在一个新添加的星号。这是因为，尽管这个提交的内容和 “C3” 完全一样，但是它实际上是一个不同的提交。这样做的原因是，它现在有一个新的源提交 C4（在最初创建 C3 时的源提交是 C1）。

一个提交仅仅包括很少的属性，比如作者，日期，变动和谁是它的父提交。如果改变其中任何一个信息，就必须创建一个全新的提交。当然，新的提交也会拥有一个新的 hash ID 。

如果还仅仅只是操作那些尚未发布的提交，重写历史记录本身也没有什么很大的问题。但是如果你重写了已经发布到公共服务器上的提交历史，这样做就非常危险了。其他的开发人员可能这时已经在最原始的提交 C3 上开始工作，并使它成为了一些新提交中不可或缺的部分，而现在你却把 C3 的改动设置到了另一个时间点（就是那个新的 C3*）。除此之外，通过 rebase 操作，这个原始的 C3 还被删除掉了，这将是非常可怕的……

因此你应该只使用 rebase 来清理你的本地工作，千万不要尝试着对那些已经被发布的提交进行这个操作。

子模块

在项目开发时，你有可能经常性地想要去引用一些库文件或其它资源文件。手动的方法就是直接下载那些必要的代码文件，然后拷贝到你的项目中，最后将这些新的文件提交到你的 Git 仓库中去。

虽然这是一种有效的方法，但是这种操作并不是最简单有效的。如果只是任意地将这些库文件提交到你的项目中，将带来一系列的问题：

- 外部代码和自己开发的代码会被合并保存在一个项目中。其实那些库文件自身就应该是一个项目，并且也应该独立于我们的工作之外。在我们当前项目的版本控制系统中，它们并不需要被保存。
- 如果库文件发生了变化（可能因为修复错误或是添加新的功能），更新这些库文件的代码对我们来说会是很繁琐的事。我们需要再次下载它的原代码文件，并且替换掉在仓库中已有的文件。

由于这些都是在日常项目开发时非常普遍存在的问题，所以 Git 也提供了一个解决方案：子模块（Submodule）。

仓库包含其它的仓库

一个 “子模块” 其实就是一个标准的 Git 仓库。不同的是，它被包含在另一个主项目的仓库中。一般情况下，它包含一些库文件和其它资源文件，你可以简单地把这些库文件作为一个子模块添加到你的主项目中。

一个子模块也是一个功能齐全的 Git 仓库，就内部而言它和别的仓库没有什么区别，你可以对它进行修改、提交、抓取、推送等等操作。

让我们来看看在实际操作中子模块是如何工作的吧。

添加一个子模块

在这个简单的项目中，我们建立一个新的 “lib” 文件目录用来存放一些库文件。

```
$ mkdir lib
$ cd lib
```

使用 “git submodule add” 命令，我们会从 GitHub 中添加一个小的 Javascript 库：

```
$ git submodule add https://github.com/djyde/ToProgress
```

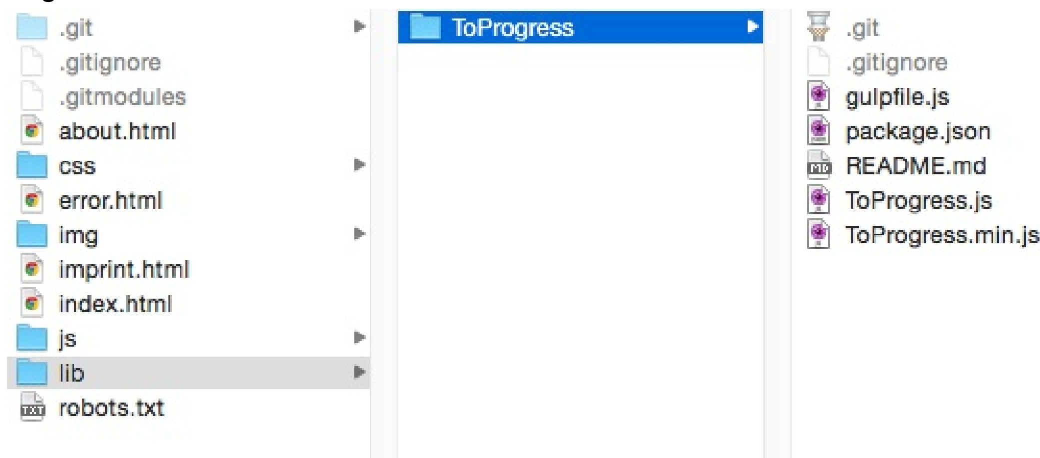
来让我们来看看现在发生了什么：

- (1) 这个命令将对一个指定的 Git 仓库进行了一个简单地克隆操作：

```
Cloning into 'lib/ToProgress'...
remote: Counting objects: 180, done.
remote: Compressing objects: 100% (89/89), done.
remote: Total 180 (delta 51), reused 0 (delta 0), pack-reused 91
Receiving objects: 100% (180/180), 29.99 KiB | 0 bytes/s, done.
Resolving deltas: 100% (90/90), done.
Checking connectivity... done.
```

- (2) 当然这一切也都会反映在我们当前项目的文件结构上。在项目中的 “lib” 目录中包括了一个新的 “ToProgess” 文件目录。通过这个文件目录所包含的 “.git” 子文件夹我们就能确认，这就是一个标准的 Git 仓库。

  <figure>

  </figure>

概念

必须要再次阐述一下：一个子模块的内容并不保存在它的父仓库中。其实只有它的远程 URL 会被记录在父仓库中，以及它在主项目中的本地路径和签出的版本。

当然，子模块的工作文件都放置在你项目的指定的目录中。最后当你要使用这些库文件时，你会发现它们并不是主项目的版本控制的一部分。

- (3) 一个新的 “.gitmodules” 文件会被创建。这个文件就是 Git 用来跟踪我们的子模块并保存它的配置信息的：

```
[submodule "lib/ToProgress"]
    path = lib/ToProgress
    url = https://github.com/djyde/ToProgress
```

- (4) 你可能会对 Git 的内部工作原理感兴趣。除了 “.gitmodules” 配置文件，Git 也会在你本地的 “.git/config” 文件中保存对子模块的记录。最终它也会在它的 “.git/modules” 目录中保存每一个子模块的 “.git” 仓库。

概念

Git 内部对子模块的管理是非常复杂的，就像你已经看到的那些 “.gitmodules”，“.git/config”，和 “.git/modules” 等等的条目那样。因此，这里强烈不建议你去手动地修改这些配置文件。为了安全起见，一定要使用适当的 Git 命令来操作子模块。

现在让我们来看看当前的项目状态：

```
$ git status
On branch master
Changes to be committed:
    (use "git reset HEAD <file>..." to unstage)

    new file:   .gitmodules
    new file:   lib/ToProgress</file>
```

像任何其他修改一样， Git 添加了一个子模块，并且要求你提交这个改动到仓库中：

```
$ git commit -m "Add 'ToProgress' Javascript library as Submodule"
```

现在，我们已经成功地添加了一个子模块到我们主项目中来了！在了解几个不同的案例之前，让我们先来看看如何克隆一个已经包括了若干子模块的项目。

克隆一个项目和它的子模块

你已经知道了，一个项目仓库并不包含子模块的文件。主项目仓库仅仅保存子模块的配置信息来作为版本管理的一部分。 这就表示，当你要克隆一个带有子模块的项目时，在默认的情况下 “git clone” 命令仅仅接收这个项目本身。我们的 “lib” 只是一个空目录，里面没有任何文件。

你有两个选择去设置这个 “lib” 目录（或者是任何一个你保存的其他子模块，“lib” 在这里只是一个例子)：

- (a) 你可以通过将 “--recurse-submodules” 参数加在 “git clone” 上，从而让 Git 知道，当克隆完成的时候要去初始化所有的子模块。
- (b) 如果你仅仅只是简单地使用了 “git clone” 命令，并没有附带任何参数，你就需要在完成之后通过 “git submodule update --init --recursive” 命令来初始化那些子模块。

签出一个版本

一个 Git 仓库可以保存无限多个提交版本，但是仅仅只有一个文件版本能保存在你当前的工作副本中。就像任何其他的 Git 仓库一样，你必须自己来决定在子模块上的哪一个版本应该被签出到你的工作副本中。

概念

和一个普通的 Git 仓库不一样的是，子模块永远指向一个特定的提交，而不是分支。这是因为一个分支的内容可以在任何时间通过新的提交来改变。所以指向一个特定的提交版本就能始终保证代码的正确。

比方说，我们希望在我们项目中使用一个旧版本的 “ToProgress” 库。首先，我需要看一下这个库的提交历史记录。我们需要切换到这个子模块的根目录下，然后执行 “log” 命令：

```
$ cd lib/ToProgress/
$ git log --oneline --decorate
```

在我们来检查实际的历史记录之前，有一点我想强调一下：Git 命令是对上下文环境很敏感的！也就是说，通过命令行来切换到子模块的目录后，我们执行的所有 Git 命令仅仅只会对子模块有效，而不是对它的父仓库。

现在历史记录被打印出来了，我们会发现这个提交被标记成了 “0.1.1”：

```
83298f7 (HEAD, master) update .gitignore
a3b6186 remove page
ed693b7 update doc
3557a0e (tag: 0.1.1) change version code
2421796 update readme
```

这就是我们希望在我们的项目使用的版本。首先我们可以来简单地看看这个提交：

```
$ git checkout 0.1.1
```

再让我们来看看父仓库。在主项目的目录中执行下面的命令：

```
$ git submodule status
+3557a0e0f7280fb3aba18fb9035d204c7de6344f   lib/ToProgress (0.1.1)
```

通过使用 “git submodule status”，我们可以查看子模块的哪一个版本在当前被签出了。在 hash 之前的 “+” 符号是非常重要的，它表明该子模块在它父仓库的官方记录中存在一个不同的版本。这是合理的，因为我们的确修改并签出了版本标记为 “0.1.1” 的提交。

如果在父仓库上执行 “git status” 命令，我们就会发现像任何其他的变化一样，Git 移动了指向这个子模块的指针：

```
$ git status
On branch master
Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   lib/ToProgress (new commits)</file> </file>
```

为了使这个改动生效，我们现在需要提交它到仓库中：

```
$ git commit -a -m "Moved Submodule pointer to version 1.1.0"
```

更新一个子模块，当指向它的指针发生了变化之后

我们看到了如何签出一个子模块的特定版本。但是，如果是开发团队的其他成员在项目中改变了对子模块的指针呢？当他移动了指向子模块的指针到另一个版本之后，我们就要整合他的改动，例如通过抓取，合并，或是 rebase ：

```
$ git pull
Updating 43d0c47..3919c52
Fast-forward
 lib/ToProgress | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Git 会以一个相当含蓄的方式告诉我们，“lib/ToProgress” 发生了变化。再次使用 “git submodule” 命令来索取更多和更细节的信息：

```
$ git submodule status
+83298f72c975c29f727c846579c297938492b245 lib/ToProgress (0.1.1-8-g83298f7)
```

还记得那个小的 “+” 符号吗？这表明子模块发生了变化，我们当前签出的子模块版本不是主项目使用的中的 “官方” 版本。

使用 “update” 命令可以帮助我们修正它：

```
$ git submodule update lib/ToProgress
Submodule path 'lib/ToProgress': checked out '3557a0e0f7280fb3aba18fb9035d204c7de6344f'
```

注释

在大多数情况下，使用 “git submodule” 家族的命令是不需要指定一个特定子模块的。但是正如上面的例子一样，如果我们给出一个子模块的路径，这个操作就只会针对那个给定的子模块。

现在我们签出了相同版本的子模块，这就是之前另一个团队成员提交到项目中的那个。

值得注意的是，“update” 命令会为你下载子模块的改动。设想一下，你的队友在你之前已经改变了指向子模块版本的指针。在这种情况下，Git 会为你获取在子模块的相应版本，并且签出这个子模块的版本，非常方便。

检查子模块的最新变化

正常情况下，你是不会经常改变库文件的代码的。如果这个子模块被真正地测试过，并且你也知道它非常匹配你的开发项目时，你才会使用它。 无论如何，子模块功能最大优点之一就是你可以很方便地与最新的发行版本同步（也许只是同步一个小小的优化）。

让我们来看看子模块是否提供了新的代码版本：

```
$ cd lib/ToProgress
   $ git fetch
   remote: Counting objects: 3, done.
   remote: Compressing objects: 100% (3/3), done.
   remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
   Unpacking objects: 100% (3/3), done.
   From https://github.com/djyde/ToProgress
       83298f7..3e20bc2  master     -> origin/master
```

请注意！现在我们切换到了子模块的文件夹，之后的操作就像对待任何一个普通的项目仓库一样（因为它就是一个普通的 Git 仓库）。 现在 “git fetch” 命令显示，当前的确存在一些新的改动在子模块的远程上。

概念

在我们准备整合这些改动之前，我想再次重申一下。当检查这个子模块的状态时，我们会发现我们正处在一个**detached HEAD**上：

```
$ git status
    HEAD detached at 3557a0e
    nothing to commit, working directory clean
```

一般情况下，在 Git 中你总是会签出某个分支。然而你也可以选择签出一个特定的提交（而不是一个分支）。这是一种比较罕见的情况，在 Git 中通常应该避免。

然而在子模块上工作时，签出某个提交的情况是非常正常的。你要确保在你的项目中，签出一个确切的静态的提交（不是一个分支），并转移到一个较新的提交上。

现在让我们通过拉取操作来整合那些新的改动到你的本地子模块仓库中吧。请注意！你不能使用那个简写的 “git pull” 命令语法，而是需要指定特定的远程和分支。

这是因为我们正处于 “detached HEAD” 状态。因为在这个时刻你不是在本地分支上，你需要告诉 Git，你想要把拉取出来的改动整合到哪一个分支上去。

```
$ git pull origin master
```

如果你现在已经执行过一遍 “git status” 命令了，你会发现我们的状态仍然处于 detached HEAD，并且在同一个提交上，当前被签出的内容并没有发生改变。如果我们在项目中想要使用这个升级后的子模块的代码，我们必须明确地将 HEAD 指针移动到分支上：

```
$ git checkout master
```

我们已经完成了在子模块上的工作，现在让我们切换回我们主项目吧：

```
$ cd ../..
$ git submodule status
+3e20bc25457aa56bdb243c0e5c77549ea0a6a927 lib/ToProgress (0.1.1-9-g3e20bc2)
```

由于我们刚刚移动了子模块的指针到了一个不同的版本，我们需要将这个改动提交到父仓库中去，从而让它成为主项目当前正式引用的“官方”版本。

在子模块中工作

有些时候，你可能会想要在子模块中作一些自己的改动。你已经知道了在子模块中工作就和在一个普通的 Git 仓库中工作一样，你在子模块目录中执行的所有的 Git 命令只会对这个子模块仓库有效。

比方说，你想对子模块进行一个小小的改动，你编辑了相关的文件，把它们添加到暂存区，并且提交它。

现在你可能会踩到第一块香蕉皮。因为如果当前你正处于一个 detached HEAD 状态，你的提交会迷失方向，它并没有关联到任何一个分支。一旦你签出了其他的东西，它的内容就会丢失。所以你应该在提交之前确保，你当前已经在子模块中签出了一个分支。

除此之外，你已经学到的其它一切 Git 操作都仍然适用。在主项目中 “git submodule status” 会告诉你指向该子模块的指针被移动了，你必须提交这个改动。

顺便提一下，如果你的子模块中还有未提交的改动，Git 也会在主项目中提醒你：

```
$ git status
...
    modified:   lib/ToProgress (modified content)
```

请务必始终保持子模块有一个干净的状态。

删除一个子模块

尽管很少会从项目中删除一个子模块，但是如果你确定想要这么做，也请不要手动地删除它，一旦所的有配置文件被打乱，将会不可避免地导致出现一系列问题。

```
$ git submodule deinit lib/ToProgress
$ git rm lib/ToPogress
$ git status
...
    modified:   .gitmodules
    deleted:    lib/ToProgress
```

使用 “git submodule deinit”，我们可以确保从配置文件中完全地删除一个子模块。 使用 “git rm” ，我们可以最终删除这个子模块的文件，包括一些其它废弃的部分。

提交这些改动，这个子模块就会从你的项目中被彻底地删除了。

git-flow 的工作流程

当在团队开发中使用版本控制系统时，商定一个统一的工作流程是至关重要的。Git 的确可以在各个方面做很多事情，然而，如果在你的团队中还没有能形成一个特定有效的工作流程，那么混乱就将是不可避免的。

基本上你可以定义一个完全适合你自己项目的工作流程，或者使用一个别人定义好的。

在这章节中我们将一起学习一个当前非常流行的工作流程 git-flow。

什么是 git-flow？

一旦安装安装 git-flow，你将会拥有一些扩展命令。这些命令会在一个预定义的顺序下自动执行多个操作。是的，这就是我们的工作流程！

git-flow 并不是要替代 Git，它仅仅是非常聪明有效地把标准的 Git 命令用脚本组合了起来。

严格来讲，你并不需要安装什么特别的东西就可以使用 git-flow 工作流程。你只需要了解，哪些工作流程是由哪些单独的任务所组成的，并且附带上正确的参数，以及在一个正确的顺序下简单执行那些对应的 Git 命令就可以了。当然，如果你使用 git-flow 脚本就会更加方便了，你就不需要把这些命令和顺序都记在脑子里。

安装 git-flow

近些年来出现了很多不同的安装方法。在本章节中我们会使用当前最流行的一种： AVH Edition。

要了解安装 git-flow 细节，请阅读下面这个文档 official documentation。

在项目中设置 git-flow

当你想把你的项目 “切换” 到 git-flow 上后，Git 还是可以像往常一样工作的。这完全是取决于你在仓库上使用特殊的 git-flow 命令或是普通的 Git 命令。换句话说，git-flow 它不会以任何一种戏剧性的方式来改变你的仓库。

话虽如此，git-flow 却存在一些限制。让我们开始在一个新的项目上初始化它吧，之后我们就会有所发现：

```
$ git flow init
Initialized empty Git repository in /Users/tobi/acme-website/.git/
Branch name for production releases: [master]
Branch name for "next release" development: [develop]

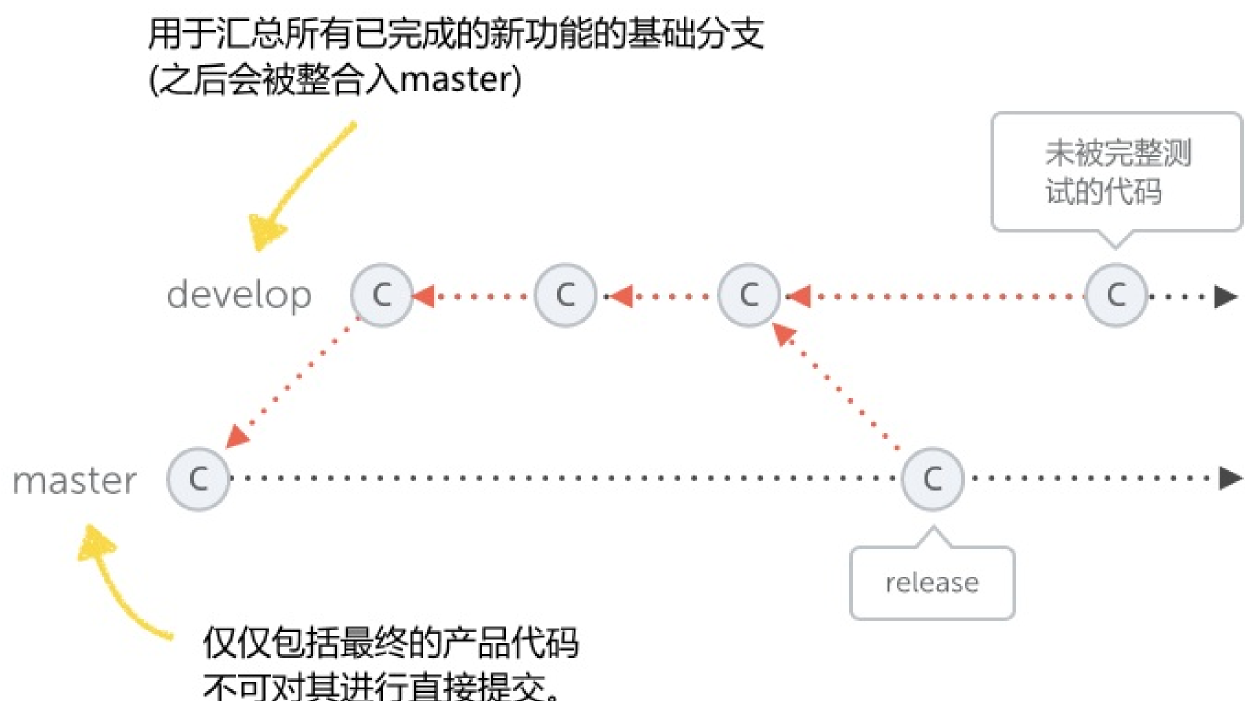
How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
```

当在项目的根目录执行 “git flow init” 命令时（它是否已经包括了一个 Git 仓库并不重要），一个交互式安装助手将引导您完成这个初始化操作。听起来是不是有点炫，但实际上它只是在你的分支上配置了一些命名规则。 尽管如此，这个安装助手还是允许你使用自己喜欢的名字。我强烈建议你使用默认的命名机制，并且一步一步地确定下去。

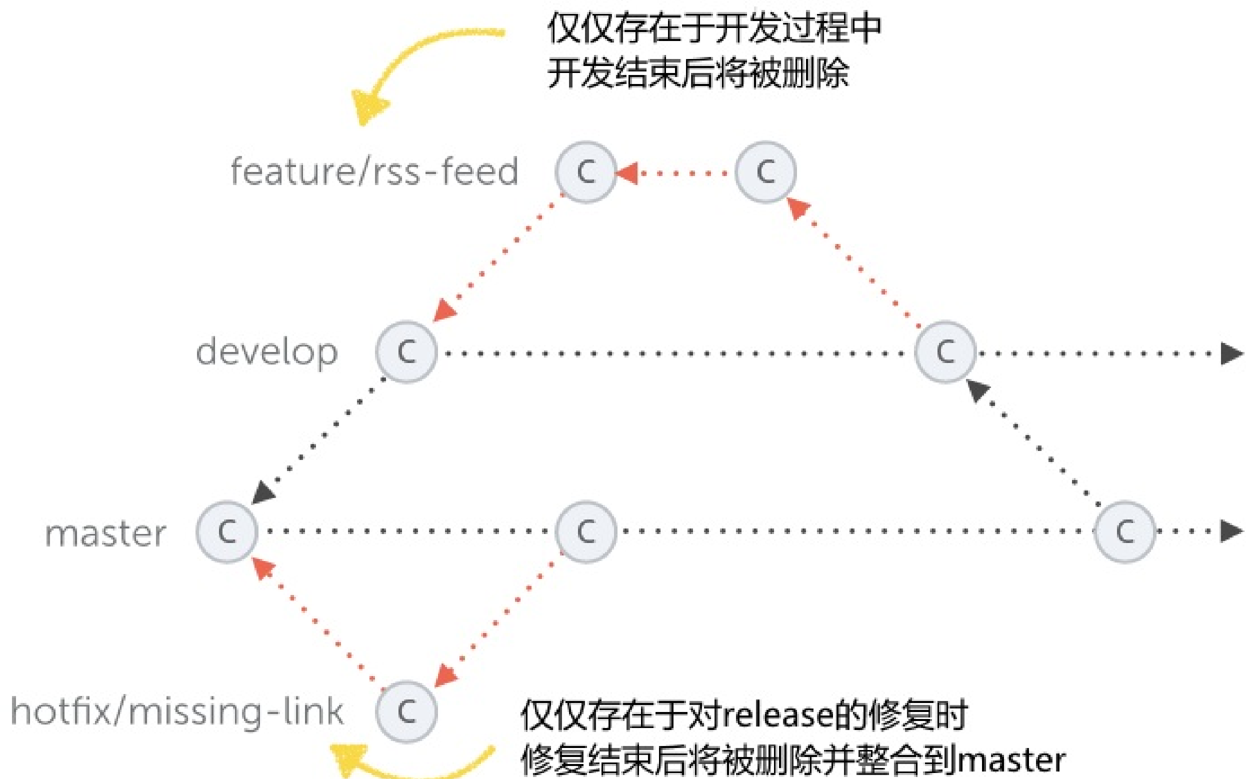
分支的模式

git-flow 模式会预设两个主分支在仓库中：

- **master** 只能用来包括产品代码。你不能直接工作在这个 master 分支上，而是在其他指定的，独立的特性分支中（这方面我们会马上谈到）。不直接提交改动到 *master* 分支上也是很多工作流程的一个共同的规则。
- **develop** 是你进行任何新的开发的基础分支。当你开始一个新的功能分支时，它将是开发的基础。另外，该分支也汇集所有已经完成的功能，并等待被整合到 *master* 分支中。



这两个分支被称作为 长期分支。它们会存活在项目的整个生命周期中。而其他的分支，例如针对功能的分支，针对发行的分支，仅仅只是临时存在的。它们是根据需要来创建的，当它们完成了自己的任务之后就会被删除掉。



让我们开始探索一些在现实应用中可能遇到的案例吧！

功能开发

对于一个开发人员来说，最平常的工作可能就是功能的开发。这就是为什么 git-flow 定义了很多对于功能开发的工作流程，从而来帮助你有组织地完成它。

开始新功能

让我们开始开发一个新功能 “rss-feed”：

```
$ git flow feature start rss-feed
Switched to a new branch 'feature/rss-feed'

Summary of actions:
- A new branch 'feature/rss-feed' was created, based on 'develop'
- You are now on branch 'feature/rss-feed'
```

概念

在这些命令的输出文本中，git-flow 会对刚刚完成的操作打印出一个很有帮助的概述。 当你需要帮助的时候，你可以随时请求帮助。例如：

```
$ git flow feature help
```

正如上面这个新功能一样，git-flow 会创建一个名为 “feature/rss-feed” 的分支（这个 “feature/” 前缀 是一个可配置的选项设置）。你已经知道了，在你做新功能开发时使用一个独立的分支是版本控制中最重要的规则之一。 git-flow 也会直接签出这个新的分支，这样你就可以直接进行工作了。

完成一个功能

经过一段时间艰苦地工作和一系列的聪明提交，我们的新功能终于完成了：

```
$ git flow feature finish rss-feed
Switched to branch 'develop'
Updating 6bcf266..41748ad
Fast-forward
    feed.xml | 0
    1 file changed, 0 insertions(+), 0 deletions(-)
    create mode 100644 feed.xml
Deleted branch feature/rss-feed (was 41748ad).
```

最重要的是，这个 “feature finish” 命令会把我们的工作整合到主 “develop” 分支中去。在这里它需要等待：

1. 一个在更广泛的 “开发” 背景下的全面测试。
2. 稍后和所有积攒在 “develop” 分支中的其它功能一起进行发布。

之后，git-flow 也会进行清理操作。它会删除这个当下已经完成的功能分支，并且换到 “develop” 分支。

管理 releases

Release 管理是版本控制处理中的另外一个非常重要的话题。让我们来看看如何利用 git-flow 创建和发布 release。

创建 release

当你认为现在在 “develop” 分支的代码已经是一个成熟的 release 版本时，这意味着：第一，它包括所有新的功能和必要的修复；第二，它已经被彻底的测试过了。如果上述两点都满足，那就是时候开始生成一个新的 release 了：

```
$ git flow release start 1.1.5
Switched to a new branch 'release/1.1.5'
```

请注意，release 分支是使用版本号命名的。这是一个明智的选择，这个命名方案还有一个很好的附带功能，那就是当我们完成了release 后，git-flow 会适当地自动去标记那些 release 提交。

有了一个 release 分支，再完成针对 release 版本号的最后准备工作（如果项目里的某些文件需要记录版本号），并且进行最后的编辑。

完成 release

现在是时候按下那个危险的红色按钮来完成我们的release了：

```
git flow release finish 1.1.5
```

这个命令会完成如下一系列的操作：

1. 首先，git-flow 会拉取远程仓库，以确保目前是最新的版本。
2. 然后，release 的内容会被合并到 “master” 和 “develop” 两个分支中去，这样不仅产品代码为最新的版本，而且新的功能分支也将基于最新代码。
3. 为便于识别和做历史参考，release 提交会被标记上这个 release 的名字（在我们的例子里是 “1.1.5”）。
4. 清理操作，版本分支会被删除，并且回到 “develop”。

从 Git 的角度来看，release 版本现在已经完成。依据你的设置，对 “master” 的提交可能已经触发了你所定义的部署流程，或者你可以通过手动部署，来让你的软件产品进入你的用户手中。

hotfix

很多时候，仅仅在几个小时或几天之后，当对 release 版本作做全面测试时，可能就会发现一些小错误。 在这种情况下，git-flow 提供一个特定的 “hotfix” 工作流程（因为在这里不管使用 “功能” 分支流程，还是 “release” 分支流程都是不恰当的）。

创建 Hotfixes

```
$ git flow hotfix start missing-link
```

这个命令会创建一个名为 “hotfix/missing-link” 的分支。因为这是对产品代码进行修复，所以这个 hotfix 分支是基于 “master” 分支。 这也是和 release 分支最明显的区别，release 分支都是基于 “develop” 分支的。因为你不应该在一个还不完全稳定的开发分支上对产品代码进行地修复。

就像 release 一样，修复这个错误当然也会直接影响到项目的版本号！

完成 Hotfixes

在把我们的修复提交到 hotfix 分支之后，就该去完成它了：

```
$ git flow hotfix finish missing-link
```

这个过程非常类似于发布一个 release 版本：

- 完成的改动会被合并到 “master” 中，同样也会合并到 “develop” 分支中，这样就可以确保这个错误不会再次出现在下一个 release 中。
- 这个 hotfix 程序将被标记起来以便于参考。
- 这个 hotfix 分支将被删除，然后切换到 “develop” 分支上去。

还是和产生 release 的流程一样，现在需要编译和部署你的产品（如果这些操作不是自动被触发的话）。

回顾一下

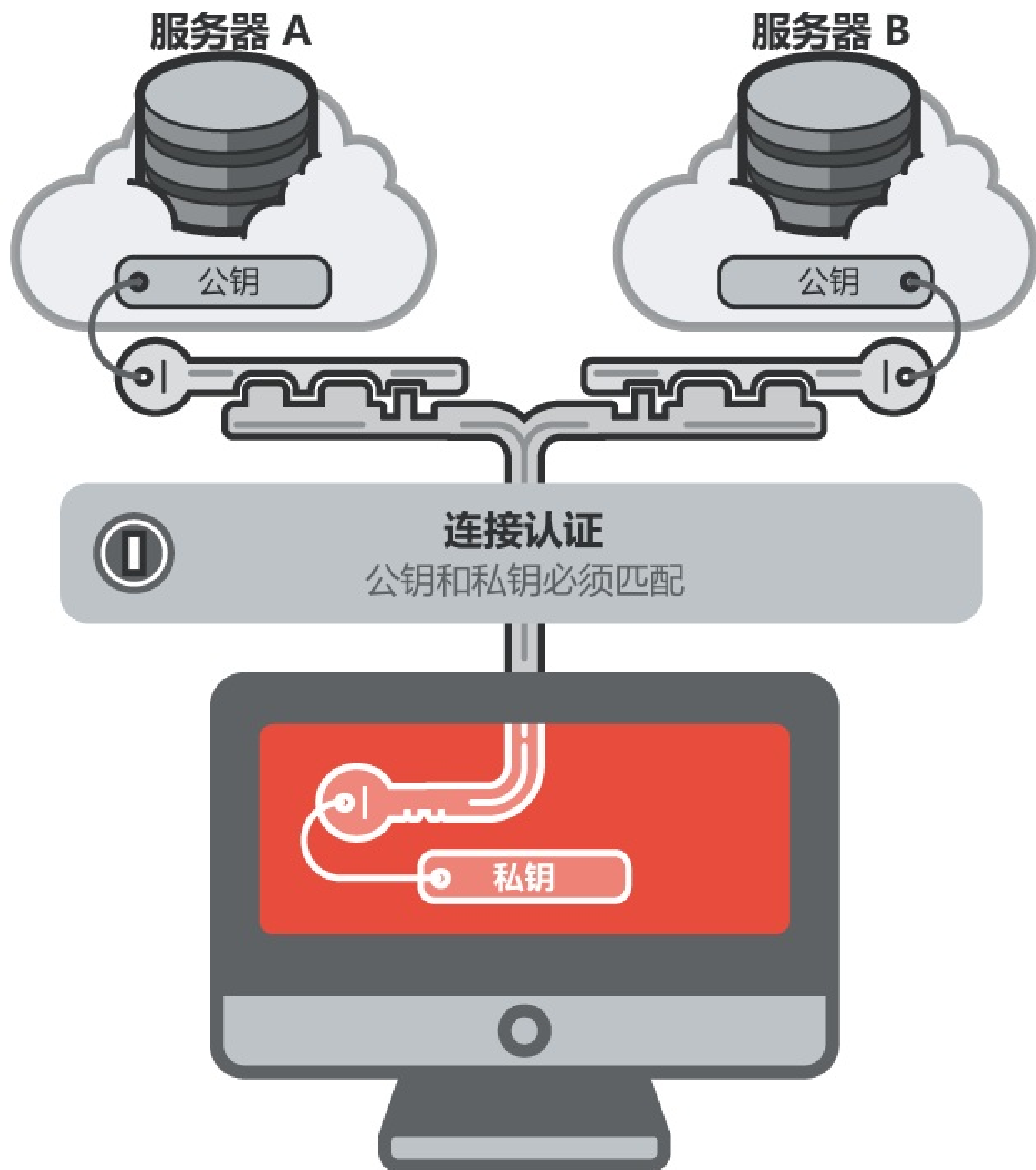
最后，在结束这个章节之前，我要再次强调几个重点。 首先，git-flow 并不会为 Git 扩展任何新的功能，它仅仅使用了脚本来捆绑了一系列 Git 命令来完成一些特定的工作流程。 其次，定义一个固定的工作流程会使得团队协作更加简单容易。无论是一个 “版本控制的新手” 还是 “Git 专家”，每一个人都知道如何来正确地完成某个任务。

记住，使用 git-flow 并不是必须的。当积攒了一定的使用经验后，很多团队会不再需要它了。当你能正确地理解工作流程的基本组成部分和目标的之后，你完全可以定义一个属于你自己的工作流程。

使用 SSH 公钥验证

通常情况下，访问远程服务器上的 Git 仓库要受到限制。你可能不希望任何人都能读取文件，或者至少不能改动文件吧。在这种情况下，进行有效的认证就是非常必要地。

你可能已经通过你所使用的浏览器了解了 “HTTPS” 协议，尽管它使用起来很简单，但是很多系统管理员还是会出于各种原因去选择使用更为普遍的 “SSH” 协议。在这种协议之下，当涉及到身份验证时，你就很可能需要 “SSH公钥”。 对于这种类型的验证需要一对密钥：一个公钥和一个私钥。私钥，顾名思义就是必须在任何情况下都保持绝对私有。它所对应的公钥则相反，应该是被安装到那些你需要登陆的服务器上。 当通过 SSH 试图建立连接时，如果客户端提供的私钥能和在服务器上所安装的公钥相匹配，那么这个客户端才会被授予访问权限。



创建一个公钥

在创建公共密钥前，你应该检查一下是否已经存在了一个：

```
$ ls ~/.ssh
```

如果在输出列表里已经存在了一个名为 “id_rsa.pub” 或是 “id_dsa.pub” 的文件，这就代表你已经有了一个密钥。在这种情况下，你可以把这个文件发给你的服务器管理员。如果你使用的是像 GitHub 或 Beanstalk 这样的托管服务，那就要把它上传到你的帐户中。

如果还没有任何密钥，你只须要执行下面这个命令来创建一个：

```
$ ssh-keygen -t rsa -C "john@example.com"
```

在 “-t” 参数之后，我们请求建立一个 “RSA” 类型的密钥。RSA 是当前最新并且最安全的一种形式。在 “-C” 参数之后，我们提供了一个注释，你可以把它想象为对这个密钥的一种描述或标签。例如使用你的 email 地址。总之，一个能让你之后更容易识别的注释。 在确认此命令后，您会被询问一些信息：

- (1) 给这个新的密钥输入一个名称，保留默认的名字和设置。
- (2) 提供一个密码。虽然 SSH 公钥可以确保在没有任何密码的情况下安全地使用，但是你还是应该设置一个密码，用来进一步提高安全性。

```
$ ssh-keygen -t rsa -C "john@example.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/tobidobi/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/tobidobi/.ssh/id_rsa.
Your public key has been saved in /Users/tobidobi/.ssh/id_rsa.pub.
The key fingerprint is:
87:23:34:de:35:d0:f2:78:05:a4:78:1b:f1:6a:7e:be john@example.com
The key's randomart image is:
+--[ RSA 2048]----+
|    . = o        |
|   ..o..         |
|  . o S .        |
| . .       . o   |
|  . + + . o      |
|   . S = + o.    |
|    . . + + . o  |
|       o .       |
|              o .|
|              .Eo|
+-----------------+
```

现在，两个密钥文件被创建出来了：“**id_rsa.pub**”（你的公钥）和 “**id_rsa**”（你的私钥）。如果你使用的是 Mac，你可以在你 home 目录下的 “.ssh” 目录中找到它们（~./ssh/）。在 Windows 中，它们应该存放在 C:\Documents and Settings\your-username.ssh\ 或是 C:\Users\your-username.ssh 中。

如果你想看看你的公钥文件的实际内容，你将会看到这些内容：

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAB3nZaC1aycAAEU+/ZdulUJoeuchOUU02/j18L7fo+ltQ0f322+Au/9yy9oaABBRCrHN/yo88BC0AB3
```

这就是公钥的内容，你需要把它安装在你所需要登陆的远程服务器上。如果你的项目开发团队拥有自己的服务器，那么你只需要把它提供给你的服务器管理员就可以了。如果你使用的是像 GitHub 或 Beanstalk 这样的托管服务，你则必须把它上传到你的帐户上。

你要把公钥的内容完全一模一样地复制出来，你可以使用下面的命令来安全方便地复制这些内容到你的剪贴板上：

```
$ pbcopy < ~/.ssh/id_rsa.pub            [on Mac]
$ clip < ~/.ssh/id_rsa.pub              [on Windows]
```

Part 5 - 工具与服务

桌面应用程序

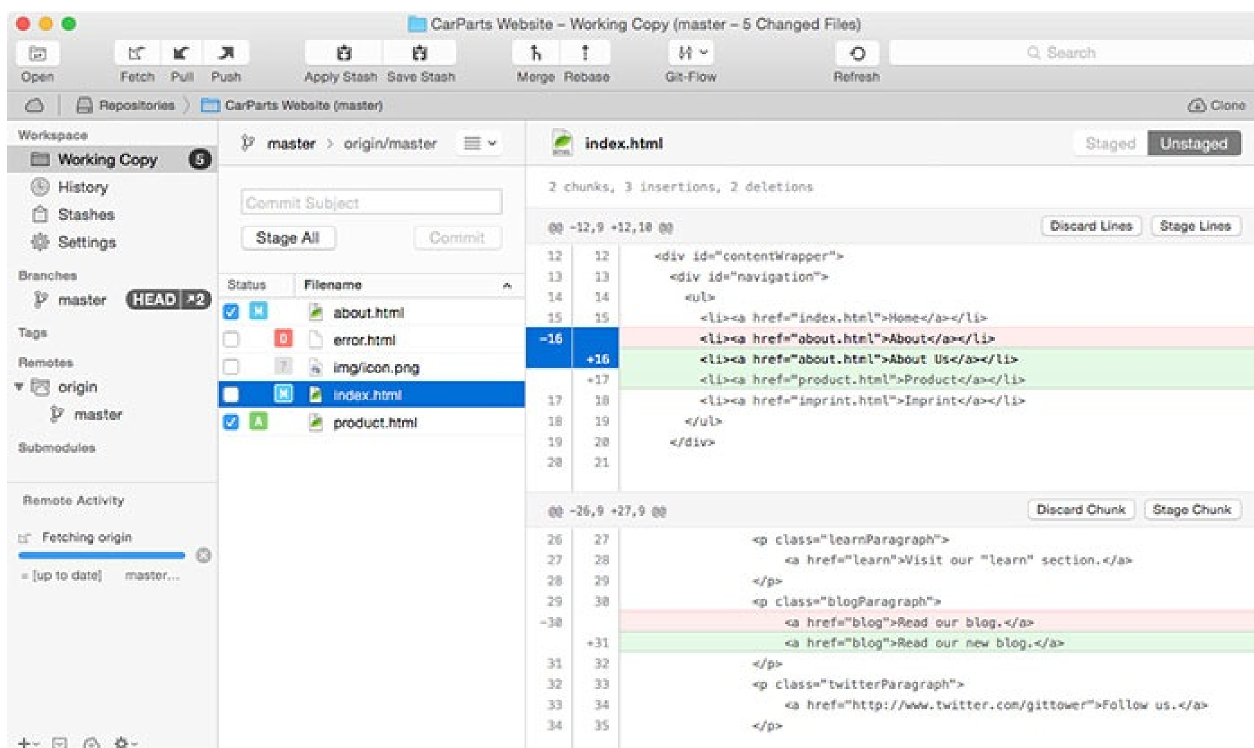
在学习这本书的过程中，你已经掌握了很多关于 Git 的命令。虽然这些是在学习过程中不可缺少的，但是版本控制的核心并不是让你学习所有的命令和参数。

当你掌握一些基本的概念，再加上一个带有用户图形界面的应用程序的帮助，就可以让你的日常工作变得更加简单。一个最大的好处就是它会为你提供了一个可视化的用户操作界面。在桌面应用程序中，很多任务使用起来会更加容易和更方便。并且你也不需要记住那几十个繁琐的 Git 命令，包括它的语法和参数。

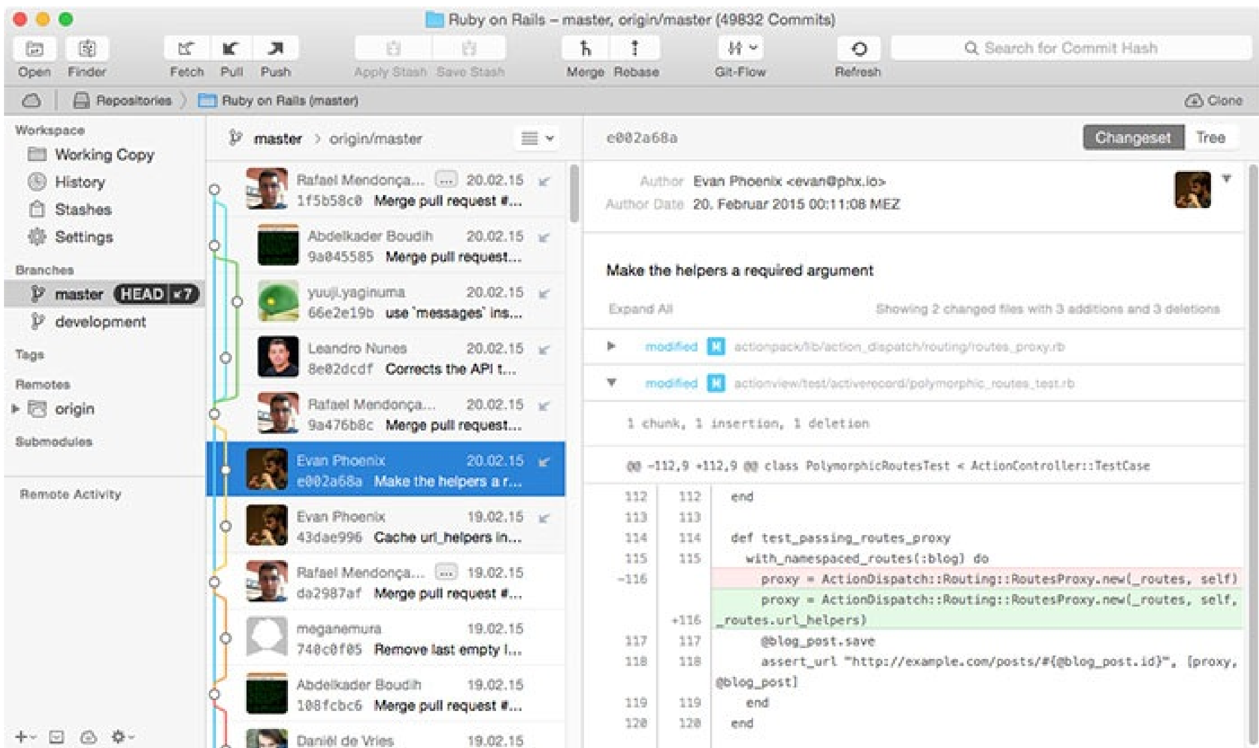
一个优秀的桌面应用程序会让你的工作更有效率，并且能够使你更有把握地运用所有 Git 提供的优秀功能。

Mac OS X

Mac 用户应该尝试一下这个程序 **Tower**。这个桌面应用程序得到了很多个人软件开发者，甚至也包括和像苹果、谷歌、亚马逊、eBay 和 Twitter 等公司的青睐。 凭借它易于使用的用户界面，大大地降低了使用 Git 的复杂性。同时它还能更为完善地展现 Git 所有的先进功能。



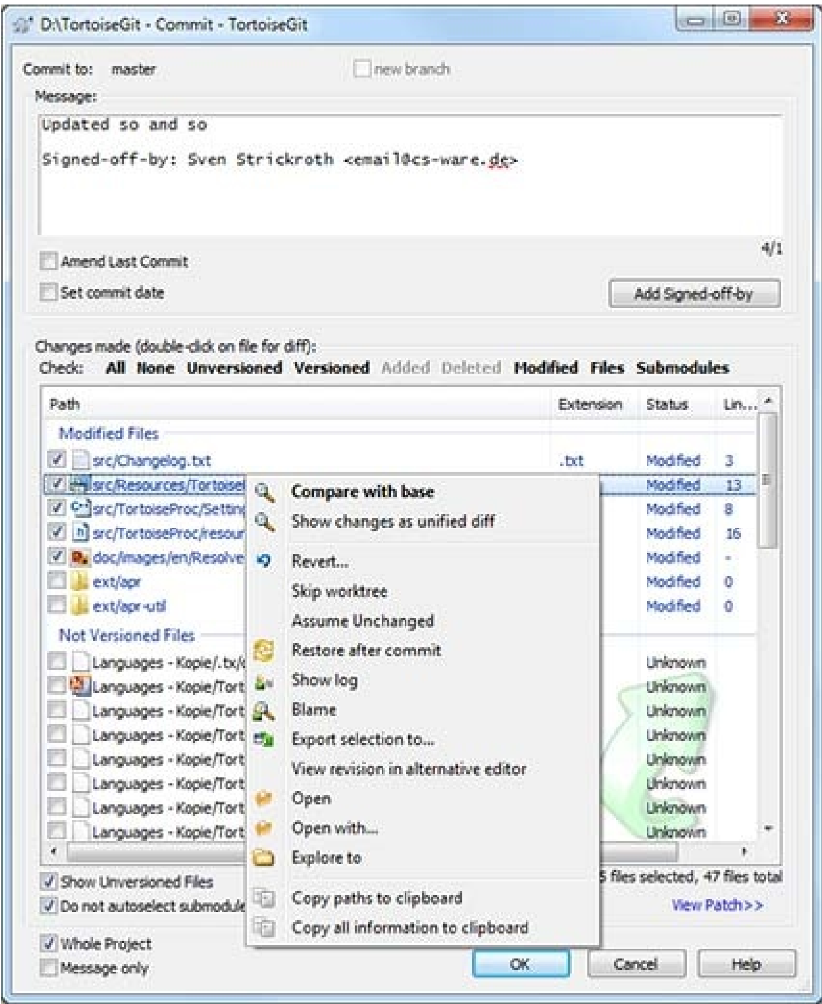
状态视图（Status View）会为你显示出所有改动过的文件，它们发生了一个什么样的改动，以及哪些文件被暂存到下一个提交中了。



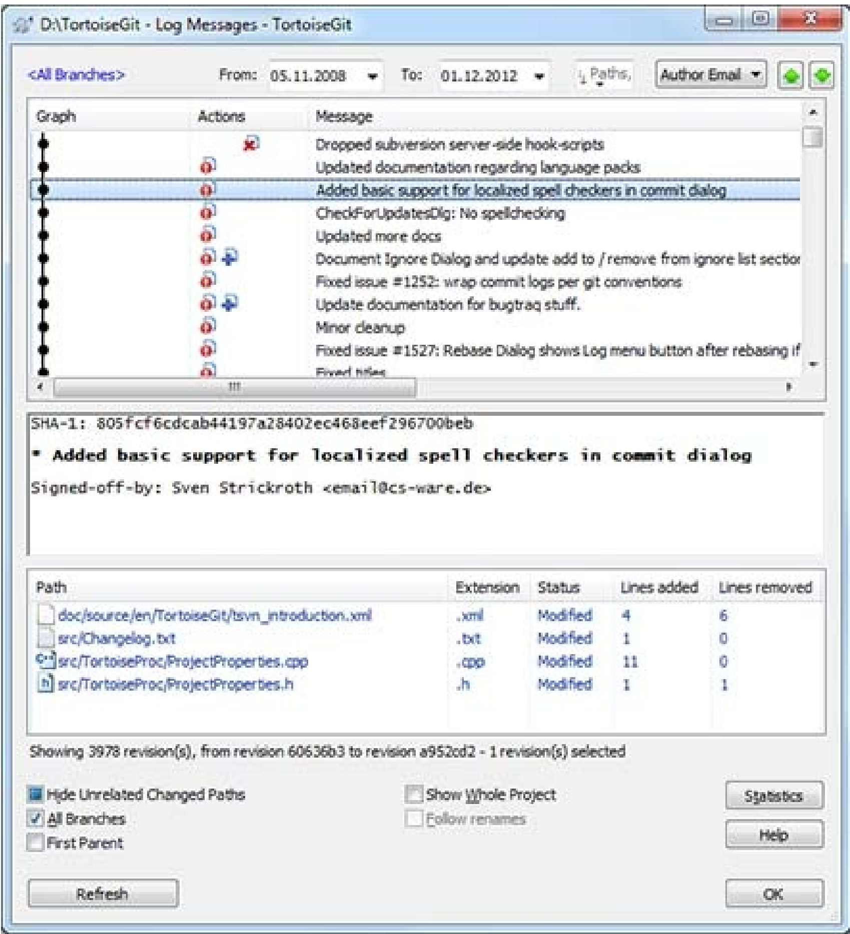
历史视图（history view）使用了一个经典的“邮箱列表形式”为你呈现出所有的提交。界面的下半部分同时也会为你显示出这个提交的详细信息，例如那些被整合文件的差异信息。

Windows

Mac 用户可以看一下这个 Tortoise Git。



那些使用过 “Tortoise SVN” 的用户应该会熟悉这个应用程序。



所有的基本功能都可以很快速地上手。

比较和整合工具

要了解在项目中都发生了什么，你就需要检查发生的改动。由于改动过的文件会被表示为“diff”，所以关键就是在于更好地理解这些改动差异。

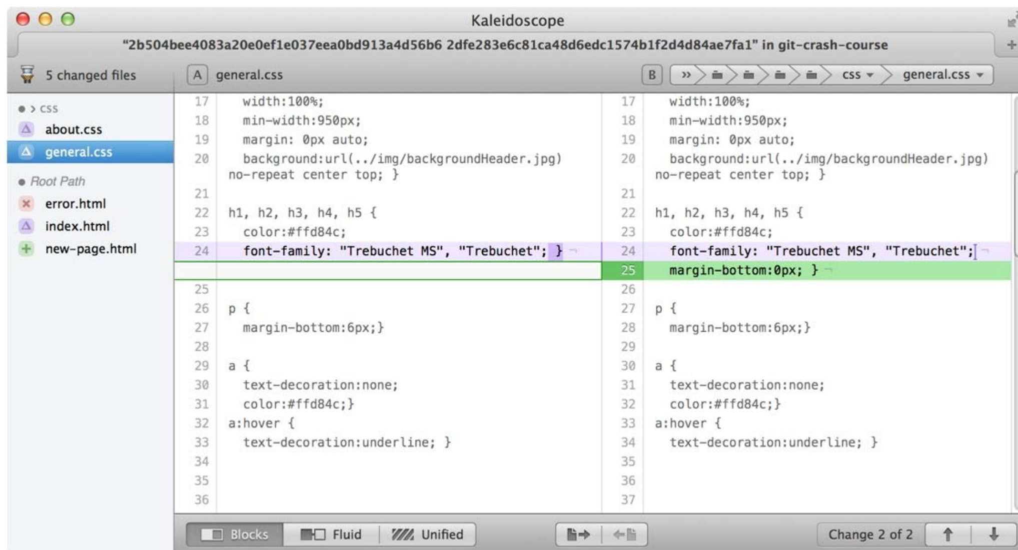
尽管在命令行界面中可以非常简单地输出这些差异信息，但是要读懂这些信息还是很有难度的。

```
$ git diff
diff --git a/css/about.css b/css/about.css
index e69de29..4b5800f 100644
--- a/css/about.css
+++ b/css/about.css
@@ -0,0 +1,2 @@
+h1 {
+  line-height:30px; }
\ No newline at end of file
diff --git a/css/general.css b/css/general.css
index a3b8935..d472b7f 100644
--- a/css/general.css
+++ b/css/general.css
@@ -21,7 +21,8 @@ body {

 h1, h2, h3, h4, h5 {
   color:#ffd84c;
-  font-family: "Trebuchet MS", "Trebuchet"; }
+  font-family: "Trebuchet MS", "Trebuchet";
+  margin-bottom:0px; }

 p {
   margin-bottom:6px;}
diff --git a/error.html b/error.html
deleted file mode 100644
index 78alc33..0000000
--- a/error.html
+++ /dev/null
@@ -1,43 +0,0 @@
- <html>
-
-   <head>
-     <title>Tower :: Imprint</title>
-     <link rel="shortcut icon" href="img/favicon.ico" />
-     <link type="text/css" href="css/general.css" />
-   </head>
-
```

比较工具仅仅致力于一个单一的工作，那就是帮助你来更容易地理解这些差异。它会使用不同的颜色，特殊的格式，甚至是不同的布局（并排，组合单个列等等）来展现出不同文件中差异：



其中一些工具甚至还可以帮助你解决合并冲突。特别是在这种情况下，你很快就能体会到，它可以非常有效地帮助降低操作的复杂性并避免错误的产生。

今天，我们可以找到很多强大的比较工具。下面这个列表就为你列出了一些：

Mac OS X

- Kaleidoscope www.kaleidoscopeapp.com
- Araxis Merge: www.araxis.com
- DeltaWalker: www.deltopia.com

Windows

- Beyond Compare: www.scootersoftware.com
- Araxis Merge: www.araxis.com
- P4Merge: www.perforce.com

代码托管服务

当你想要分享你代码给其他人，或者是你需要在另外一台电脑上工作时，托管代码就会是一个非常重要的话题。基本上代码托管有两种不同的形式： do-it-yourself（建立一个自己的）或者 leave-me-in-peace（使用第三方提供的平台，不麻烦自己)。

(A) Do-It-Yourself

把你的 Git 仓库托管在你自己的服务器上会有很多的优点：

- 可以节省你花在代码托管服务上的费用。
- 你的代码只保存在你自己的服务器内部。
- 你会有很多自由发挥的空间。

当然这里也存在一些缺点：

- 你必须要保证服务器的正常的运行时间，用以确保它的可用性。
- 你必须要负责进行备份工作（这是一个非常重要且繁琐的工作）。
- 你必须要负责它的安全维护和更新。

最后，托管代码的最艰巨的任务并不是去管理那些 Git 仓库，而是对服务器本身的管理和维护。不要误会我的意思，我的本意并不是“不要自己托管自己代码，而去使用一个现成代码托管服务”。我的本意是“让你明白代码托管的真正含义”。 如果你有足够的经验和能力去完成它，那么托管你的代码仓库到你自己的服务器上会是最好的选择！

(B) Leave-Me-In-Peace

对于大多数人来说，他们并不具备维护服务器的能力。虽然很多人都或多或少的地掌握一些理论上的知识，但是这还远远谈不上精通。 现在你可以找到几十个专门的代码托管服务供应商，它们可以为你提供比如服务器管理，备份，安全维护等等全方位的服务。为了让你能快速地了解这方面的信息，我们为你整理出了一个简短的列表。

GitHub www.github.com

GitHub 是在 Git 的世界中最流行的代码托管服务。特别是对于开源项目， GitHub 是最值得推荐的平台。

Beanstalk www.beanstalkapp.com

Beanstalk 不仅仅提供 Git 仓库的托管，而且它还支持 Subversion 项目。作为一个非常精简和可靠的服务供应商，Beanstalk 是企业级用户的最佳选择。

Bitbucket www.bitbucket.com

除了对 Git 仓库支持外， Bitbucket 也同时支持对 Mercurial VCS 的托管。它有着和 GitHub 平台很类似的功能，但是在开放源代码世界中它并不像 GitHub 那样受欢迎。

Plan.io www.plan.io

Plan.io 提供了一个完整的项目管理平台。除了支持对 Git 和 Subversion 的代码托管之外，它还提供了模块化的任务管理，客户服务支持，甚至还集成了Wiki。

更多学习资源

近些年来出现了大量的关于 Git 的文档，教程和文章。我建议你浏览一下这些在线资源：

- 命令速查表
- "Git - the Simple Guide"
- "Pro Git" ebook

附录

版本控制的最佳实践

提交对映改动

一次提交要包括一个相关改动。例如，对于两个错误的修复应该进行两次不同的提交。精简的提交可以让其他的开发团队人员更简单地明白其改动的用义。如果其中一次提交的改动出现了问题，也可以方便地回滚到改动之前的状态。借助暂存功能来标记相关的改动文件，Git可以为你打造出非常精准的提交。

频繁地提交改动

经常性地提交改动可以确保不会出现特别庞大的提交，同时也可以比较精准地对应到所需要的改动上。此外，通过频繁地提交也可以比较快速地和其他开发人员来共享你的改动。同样也会避免在整合代码时出现过多的合并冲突。相反的，非常庞大的提交会加大整合代码时出现冲突的风险，解决这些冲突也会非常复杂。

不要提交不完整的改动

虽然原则上来说不要提交一些还没有完成的改动，但是对于一个非常庞大的新功能来说，也并不意味着你必须整体完成这个功能后才可以提交。恰恰相反，你必须把那些改动正确地分割成一些有意义的逻辑模块来进行频繁地提交。如果你仅仅是因为急着想要下班，或者是想要得到一个干净的工作副本（比如想要切换到另一个分支上），你可以利用 Git 所提供的储藏（Stash）功能来解决这些问题。切记不要把那些不完整的改动提交到仓库中。

提交前测试那些改动

不要理所当然地认为自己完成的改动都是正确的。所有的改动一定要通过彻底地测试才表示它真正地被完成了。尽管这些改动可能仅仅是提交到了你的本地仓库中，只有你自己才能看到，但完整的测试同样是非常重要的，因为这些代码可能之后会被推送和共享到远程给其他的开发人员。

高质量的提交注释

提交注释的标题需要一个少于50个字符的简短说明。在一个空白的分割行之后要对改动的细节进行一个详细地描述。例如尝试着回答两个问题：出于什么原因需要进行这次修改？具体改动了些什么？为了和自动生成的提交注释保持一致（例如 git merge 可能会自动生成提

交），一定要使用现在时祈使句（例如要使用 change ，而不要使用 changed 和 changes）。

版本控制不是备份系统

版本控制系统具有一个很强大的附带功能，那就是服务器端的备份功能。但是千万不要把 VCS 仅仅当成一个备份系统。特别需要注意的是，只能提交那些有意义的改动。VCS 不是用来备份文件用的。（请参阅 <提交对映改动>）

使用分支功能

分支是 Git 一个非常强大的功能，当然这不是偶然的。自始至终，Git 的宗旨就是提供一个即快速又简单的分支功能。它是一个优秀的工具，并且可以帮助解决开发人员在日常工作中存在的代码冲突的问题，因此分支功能应该广泛地被运用在不同的开发主题中。比如添加新功能，修复错误，尝试新的想法等等。

遵循一个工作流程

Git 可以支持很多不同的工作流程：长期分支、功能分支、合并以及 rebase、git-flow 等等。选择什么样的开发流程要取决如下一些因素：项目开发的类型，部署模式和（可能是最重要的）开发团队成员的个人习惯。不管怎样，选择什么样的流程都需要得到所有开发成员的一致认可，并且一直遵循它。

命令 101

对于很多非专业人士来说，命令行界面（ CLI 、 Terminal、 bash 或者 shell）是高不可攀的，不敢轻易地染指。但是在能掌握一些最基本的操作之后，对这一领域的认识就会完全改变了。

打开命令行界面

在 Mac 上，最常见的应用命令行就是 “Terminal.app”。它会预装在每一个 Mac OS X 系统中。你可以在你的 “Applications” 目录中的子目录 “Utilities” 中启动它。

在 Windows 上，就是本书之前的安装指南中所提到应用程序 “Git Bash”。你可以在 windows 的开始菜单里的 “Git” 目录中找到它。

找到你自己的方法

顾名思义，命令行界面是用来执行各种命令的终端界面。你可以键入一些命令然后通过回车键来运行它。很多的命令都默认地依靠在你当前所处的位置，这里所说的 “位置” 就是指当前你所在的目录。 好的，让我们来执行一个命令来找出我们当前的位置吧：

```
$ pwd
```

你可以很容易地记住这个命令，它代就表：“**p**rint（打印） **w**orking（工作） **d**irectory（目录）”。它将会返回给你当前你所位于的本地文件夹的路径。

对于切换当前的工作目录，你可以使用 “cd” 命令（这里的 “cd” 代表了 “**c**hange（切换）**d**irectory（目录）”）。例如向上移动一个目录（进入当前目录的上一层目录），你只需要执行：

```
$ cd ..
```

移动到它下层的一个子目录，你可以执行：

```
$ cd name-of-subfolder/sub-subfolder/
```

你经常会看到一种特殊的路径符号：“~”。这个标志代表你的用户帐户的主文件目录。其实你并不需要每次都键入繁琐的用户名路径，比如 “cd /Users/your-username/projects/”，你可以简单地执行下面这个命令：

```
$ cd ~/projects/
```

另一个非常重要的命令就是 “ls”，它可以显示出当前目录中的内容。我建议你在使用这个命令时永远加上两个参数：“-l” 结构化的列表格式来输出内容；“-a” 显示出隐含文件 （这在版本控制系统中非常重要的）。显示当前目录的内容：

```
$ ls -la
```

对文件的操作

很多重要的文件操作都可以非常方便地用命令来完成。

让我们来删除一个文件：

```
$ rm path/to/file.ext
```

如果想要删除一个文件夹，你应该加上 “-r” 参数 （ r 代表了 “recursive（递归）”）：

```
$ rm -r path/to/folder
```

移动一个文件是很简单的：

```
$ mv path/to/file.ext different/path/file.ext
```

“mv” 命令也可以用来对一个文件进行重命名：

```
$ mv old-filename.ext new-filename.ext
```

假如你不是要移动这个文件而是复制它，用 “cp” 命令来替换那个 “mv” 命令就可以了。

最后，你可以使用 “make directory” 命令来创建一个目录：

```
$ mkdir new-folder
```

生成输出

命令行是无所不能的。它可以显示一个文件的内容，但是它却不可能像一个专业的文本编辑器那样方便。尽管如此，在某些时候它还是非常实用的。例如当你仅仅是想要进行一个快速的预览，或者当你正在远程服务器上工作时，GUI 应用程序并不支持的情况之下。

“cat” 命令会输出完整的文件内容：

```
$ cat file.ext
```

同样的，“head” 命令只会显示文件的前10行，“tail” 会显示文件的后10行。和其它其他应用程序一样，你可以通过简单地滚动鼠标来继续显示。

“less” 命令在这方面就有点不同了。

```
$ less file.ext
```

尽管它也可以用来显示文件内容，但是它能够控制页面流本身。也就是说，它只显示一个整页面的内容，然后等待你的明确指令。当显示的文件并不完整时，你会发现在屏幕的最后一行会显示出该文件的名称或者仅仅显示一个冒号（“:”），它会等待你的指令。敲击空格键可以向下翻页，“b” 可以向上翻页，“q” 可以退出 “less” 程序。

命令行让你的生活更容易

在用命令行工作时有一些小窍门可以让你使用起来更方便容易。

TAB 键

当你输入文件或者目录（包括它的路径），利用 TAB 键就会非常方便。它会自动地把你的输入补全，这是非常有效率的。例如，如果你想要切换到一个不同的目录，你可以键入整个路径的每个字符：

```
$ cd ~/projects/acmedesign/documentation/
```

或者你也可以利用 TAB 键（自己尝试一下吧！）：

```
$ cd ~/pr[TAB]ojects/ac[TAB]medesign/doc[TAB]umentation/
```

如果你键入的字符是不明确的（因为 “dev” 可能是 “development” 或者是 “developers” 目录……），命令行应用就不可能自动补全你的输入。在这种情况下，你可以再一次敲击 TAB 来得到所有匹配的内容，并且借此可以再键入更多的字符。

方向键

命令行界面可以保存一些你最新执行过的命令。使用键盘上的向上键，你可以一个一个地调出你刚刚使用过的命令（从最近使用过开始）。使用键盘的向下键则反之。

CTRL 键

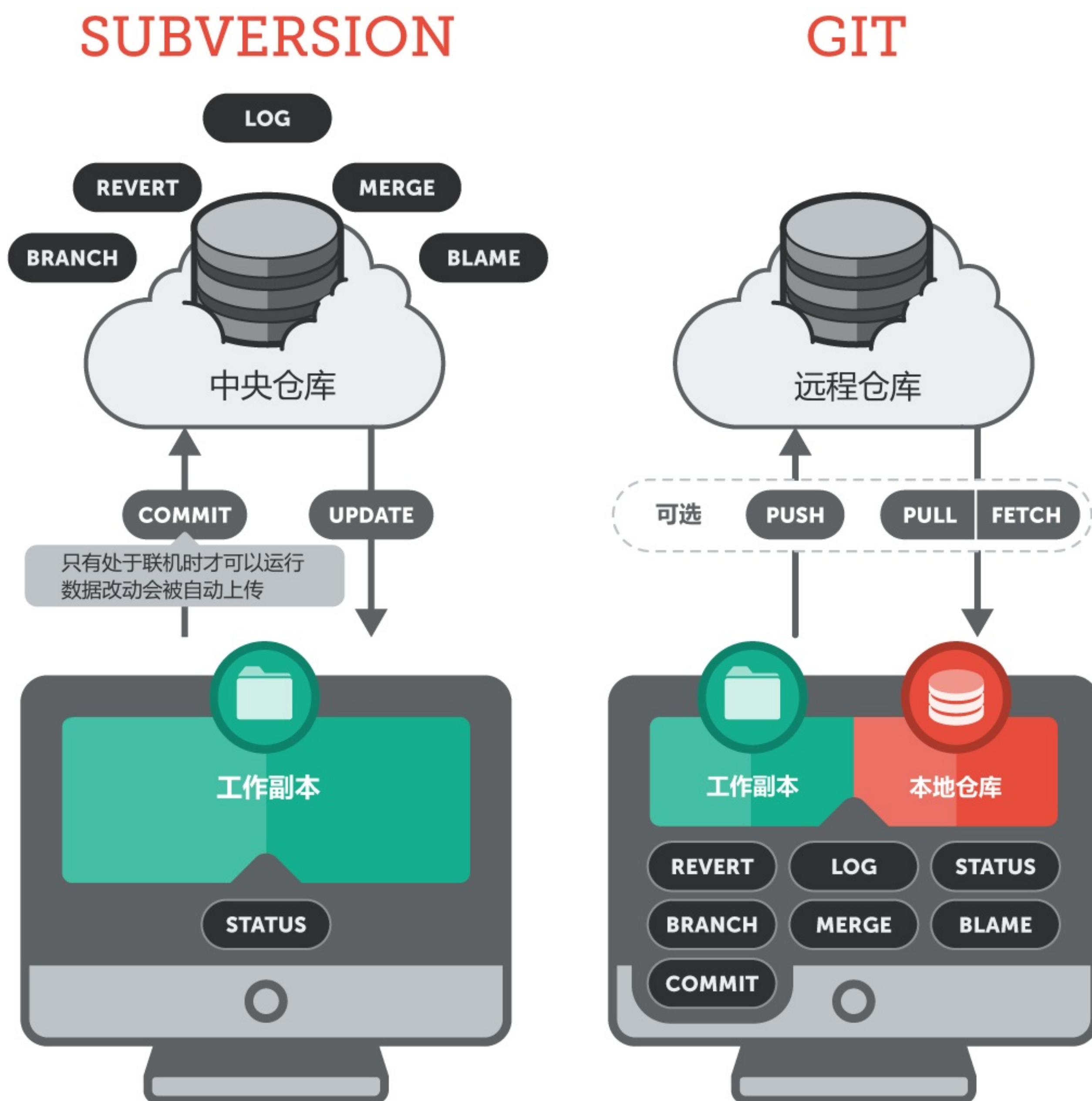
在键入命令时使用 CTRL+A 可以移动光标到行首，使用 CTRL+E 可以移动光标到行末。 然而，并不是所有的命令都是通过简单的回车就可以完成的，有些需要你的进一步的指令。如果你被卡在一个命令的中间而你希望终止它时，可以使用 CTRL+C 强行终止这个命令。在大多数情况下这是安全的。但是还是要小心，中止某些命令可能会让系统处于不稳定的状态。

从 Subversion 过渡到 Git

目前，想从 Subversion 过渡到 Git 其实并不困难，只要你不把 Git 和 Subversion 混淆就行。一旦你明白了两者在概念上的区别，这个改变的过程就会变得容易。

分布式与集中式

Subversion 是一个集中式（*centralized*）的版本控制系统。所有的开发团队成员都工作在单一的远程中央仓库上，当在这个中央仓库上进行“签出（checkout）”操作时，它就会在你的本地计算机上设置一个“工作副本（working copy）”。这就是一个存储在你本地计算机上的一个特定版本的快照。



Git 是一个分部式（*distributed*）的版本控制系统，它有着一个不同的工作方式。相对于 Subversion 的 “签出（checkout）”，每一个 Git 用户会从远程仓库 “克隆（clone）” 出一个本地仓库。反过来说，一个用户会得到一个完整的仓库，而不仅仅只是一个工作副本。用户在本地计算机上拥有自己的仓库，并且包含所有的项目历史记录。 用户可以在自己的本地计算机上做任何想要操作，例如提交（commit），历史检查（inspect history），恢复到一个旧的版本等等。只有当你想要共享你的工作结果时，你才需要连接到远程服务器上。

仓库结构和 **URLs**

一个 Subversion 的仓库通常都是由几个目录组织起来的。“trunk” 目录对应你的开发主线，“branches” 目录对应那些特定的工作背景下的开发，而 “tags” 目录则用来标记一个特定的版本。它们都要通过自己的 URL 来指向到它在中央仓库中的具体位置：

```
svn+ssh://svn@example.com/svn/trunk
```

Git 仓库就完全不一样了，它的组成完全就是一个在项目根目录下的 “.git” 文件夹。对分支和标记的查找完全依靠命令，而不是通过 URLs。Git 的 URL 只指向仓库的位置。

```
ssh://git@example.com/path/to/git-repo.git
```

分支

正如刚才提到的， Subversion 的分支仅仅是一些有特殊含义的目录。在创建一个新的分支时，你只是把项目的当前状态完完整整地拷贝到这个新的分支目录中。

Git 的分支技术是它的设计核心，因此它拥有一个完全不同的概念。一个在 Git 中的分支就是一个指向一个特定版本的指针：不拷贝任何文件；不创建任何目录；没有任何额外的操作。在 Git 中你 永远 工作在一个分支上，至少工作在那个系统默认创建的 “master” 分支上。在你的工作副本上只包括你当前的活动分支中的文件（ Git 称之为 “HEAD”）。 所有其他的版本和分支都被保存在你的本地仓库中， 并且随时都可以非常快速地恢复到一个旧的版本。 一定要记住 Git 的分布式特性：分支可以被发布到在远程服务器上，但是本地上的分支对于日常的工作更加重要。

提交

当你想要在 Subversion 中提交一个改动，有如下的一些规则：

- 你必须确保与中央仓库的连接。你不能进行离线提交。
- 提交的内容要立即存储在中央仓库中。
- 它会被分配一个递增版本号。

提交在 Git 中就是完全另外一种情况：

- 你没有必要连接到任何一个“中央”仓库，因为在你的计算机中就拥有一个完整的本地仓库。因此提交仅仅只记录在本地仓库上。它们不会自动地传递到远程仓库中，除非你自己决定共享这个改动。
- 文件的改动并不意味着它会被自动地包含在下一次提交中。你必须指明哪些改动你想要提交，并把它添加的所谓的“暂存区（Staging Area）”中。你甚至可以只对文件的部分修改或是特定的几行代码进行提交，而其他部分则稍后提交。
- “commit hashes” 替代了版本号码。由于提交都发生在开发人员的本地计算机上，你不可能给某个提交分配一个号码 #5，而另外一个分配 #6，这就产生了个问题，在分布式系统下谁是第一个提交呢？在 Git 中，每一个提交必须拥有一个唯一的ID，因此一个哈希字符串就代替了那个依次递增的版本号。

分享工作

在 Subversion 中，在提交之后，你的工作会被自动地转移到中央仓库上去。只有在你连接到这个中央服务器时你才可以进行提交。

Git 不会自动上传任何东西。你可以自己决定，你的那些分支（也可能是所有分支）需要共享给你其他的团队成员。除此之外共享工作也是十分安全的。冲突只会出现在你的本地上，它决不可能发生在远程服务器上。这会让你有信心来解决冲突，因为你不会破坏远程仓库。

为什么选择 Git

虽然市场上有几十种不同的版本控制系统，一些世界上最著名的项目（例如 Linux 内核，Ruby on Rails，或是jQuery）都选择了使用 Git 作为它们的版本控制系统。为什么它们都选择 Git 呢？

节省时间

Git 运行快速。尽管我们在这里讨论的只是运行一个命令所需要的几秒钟，但是把它累积在你的日常工作中就是一个不小的飞跃了。它可以节省那些不必要的等待时间，并且去完成其它一些有意义的工作。

离线工作

当你不能联机远程中央仓库时你该怎么工作呢？对于一个像 Subversion 或者 CVS 的集中式版本控制系统来说，如果你没有连接到中央仓库，你就不能很好的工作。如果使用 Git ，几乎所有的东西都可以简单地在你的本地机器上完成。例如进行提交，查看你的项目历史，合并或者创建分支等等。至于在哪里工作？什么时候工作？ Git 不会给你施加任何限制。

撤销错误操作

每个人都会犯错，而使用 Git 的最大好处就在于，几乎在所有的情况下你都能 “撤消” 你的错误操作。比如如果你忘记了把一个小小的改动包含进来，因此你要改正你的上个提交。又或者你想要撤销一个完整的提交，因为这个功能有可能是不必要的。当发生了很严重的错误时，你甚至可以通过恢复引用日志来让一个提交不可见。你可以放心，Git 几乎很少真正地删除数据。

可靠性高

不用担忧，你不会在 Git 中搞砸任何东西，这种感觉是不是非常好？在你的 Git 项目中的每一个团队成员都克隆了整个项目在他们的本地计算机，这个本地克隆也可以看作一个完整的项目备份。除此之外， Git 上的操作几乎都是进行数据添加，几乎从不删除数据。这意味着丢失数据或是仓库损坏的情况几乎不可能发生。

让提交更有意义

只有包含了相关的改动的提交才有意义。想象一下，如果一个提交中包括一个新添加的功能 A ，还包括功能 B 的一部分改动，并且还存在一个对错误 C 的修复。这样其他的团队成员就很难理解这个提交的意图，而且当其中的一个改动出现了错误，撤销起来也非常麻烦。利用它独一无二的“暂存区（staging area）”概念，Git 可以帮助你打造很细微和精准的提交。你可以准确地判断哪些更改将被包含在你的下一个提交中，即使只是一行改动。Git 真正提高了对版本控制的实用性。

更高的自由度

当使用 Git 工作时，你可以定义一个对项目和团队有意义的工作流程。使用 Git 也不需要其它的要求。你可以连接多个远程仓库，使用 rebase 来替代合并，或者在需要时可以使用子模块。当然，你也可以简单地像 Subversion 那样仅仅使用一个远程的集中式仓库。无论你使用什么样的工作流程，它都有各种各样的优点。

避免混乱

关注点分离可以更明确地了解事情的进程。当你工作在功能 A 上时，不应该有任何人受到你未完成的代码的影响。如果那个功能是完全没有必要的话呢？或是完成了对它的一些改动提交后，你注意到你完全错了呢？分支功能就可以解决这些问题。当然其他版本控制系统也都有分支，但是 Git 真正的把它改进地更快速，更简单了。

顺应潮流

聪明的开发人员应该顺应潮流。Git 正在被越来越多的知名公司和开源项目所使用，如 Ruby On Rails，jQuery，Perl，Debian，Linux 内核等等。拥有一个大型的用户群体是一个很大优势，因为往往会存在很多系统去推动他的发展。大量的教程，工具和服务，这让Git更加具有吸引力。